colesoft™

Z/XDC®
USER GUIDE
Z/XDC® Release V3.1 for z/OS

David B. Cole

Z/XDC®

Z/XDC® is a member of the XDC® (Extended Debugging Controller®) family of products
COPYRIGHT "Izzi Software LLC" - 2010-2026. ALL RIGHTS RESERVED

Maintenance Level CHC-2609B
Generated 09/04/2026, 8:09am

online PDF Documents

PREFACE

PROPRIETARY LEGEND

Z/XDC® and its documentation (collectively, "Product"), including copies thereof, are the property of Izzi Software DBA Colesoft ("Owner"). Use of the product is licensed from ColeSoft Marketing, Inc. ("Licensor").

The Product may be used only by those organizations that are licensed by Licensor for such use and only in the manner so licensed. The Product may not be published, reproduced, distributed or made available to third parties for any purpose without the expressed written permission of Owner or Licensor. However, a reasonable number of copies may be made of the documentation (including the copyright notices thereon) as is necessary for the legitimate use of the Product within a licensed organization ("Customer").

Except as may be otherwise expressed in a signed agreement between Licensor and Customer, Owner and Licensor make no representations or warranties, expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose, the warranty of freedom from rightful claims by way of infringement and the like, and any warranty as to accuracy.

WARNING! Z/XDC® is a powerful tool for dynamically locating and correcting malfunctions in actively executing user programs and operating system routines. Accordingly, it is inherent in its design, that unless the use of this Product is properly controlled, then under certain conditions a malicious or careless user can use the Product to alter, subvert, counterfeit, damage or otherwise disturb the normal execution of user programs or system routines including, under certain conditions, both its own and system security routines.

Therefore, even if advised of the possibility of loss or damages, under no circumstances shall Owner or Licensor be liable for any loss or damage whatsoever (including death) arising from the Product, whether such loss or damage be direct, indirect, consequential, special or otherwise. Further, neither Owner nor Licensor shall be obligated to indemnify in any manner against any person or organization for any loss of any kind or nature which the person or organization may experience, arising out of the use or misuse of the Product.

CONTACTING COLESOFT

The **Z/XDC®** family of products is marketed by **ColeSoft Marketing, Inc**. with its principal office in Charlottesville, Virginia. If you would like more information, please contact ColeSoft Marketing as follows:

Phone:	**540-456-8210**
E-Mail:	**sales@colesoft.com**
Home Page:	**www.colesoft.com**

Our Technical Support contacts are:

Phone:	**540-456-8210**
E-Mail:	**techsupt@colesoft.com**
Home Page:	**www.colesoft.com**
FTP site:	**ftp.colesoft.com**

Our Customer Services contacts are:

Phone:	**540-456-8210**
E-Mail:	**support@colesoft.com**
Home Page:	**www.colesoft.com**

Our snail mail address is:

Address: **ColeSoft Marketing, Inc.**
440 Monticello Ave Ste 1802
PMB 380764
Norfolk, Virginia 23510-2670
USA

ONLINE PRESENCE

ColeSoft Marketing maintains the following resources on the Internet:

[Home Page] ColeSoft's Home Page is www.colesoft.com. It provides the following services:

- General information about Z/XDC
- E-mail links to both Marketing, Technical Support, and Customer Services
- FTP links for uploading diagnostic information and other files to Technical Support
- Online product delivery
- Links permitting existing customers to download a full set of Z/XDC's documentation
- 24x7 self-service for temporary, short-term, license activation codes for use in D.R. tests and other emergencies
- Occasional blog posts publicizing newly implemented features and capabilities.

[YouTube] ColeSoft's YouTube page is at youtube.com/colesoftware. This is where you will find several "how to" videos describing various aspects of using ColeSoft products. This is a wonderful resource, particularly for new Customers.

TRADEMARKS

TFS™, **XDC-TFS™**, **CDF™**, **XDC-CDF™**, **FASM™**, **base/XDC™**, **c/XDC™**, **asm/XDC™** and **dump/XDC™** are trademarks of Izzi Software.

Extended Debugging Controller®, **XDC®**, and **Z/XDC®** are registered trademarks of Izzi Software.

Other brand and product names referenced in this document are trademarks or registered trademarks of their various holders. Use of their names herein is for identification purposes only.

PRODUCT MANUALS

Z/XDC's documentation consists of the manuals shown below. The manuals are provided in PDF format. All manuals can be downloaded from www.colesoft.com under Support.

Install Guide	Describes how to install Z/XDC
User Guide	Explains how to use Z/XDC
Commands Reference	Describes all of Z/XDC's commands
Messages Reference	Explains all numbered messages issued by Z/XDC
Maintenance and New Functionality Guide	Describes what is new if Z/XDC (updated constantly)
Primer	A basic tutorial to help the beginner get started with Z/XDC
Quick Reference	A comprehensive syntax reference for all of the Z/XDC commands

ADDITIONAL MANUALS

Z/XDC customers may make as many copies of this manual as they feel is necessary for the legitimate use of Z/XDC within their organization. Existing customers may download from our web site (www.colesoft.com/product-support/zxdc-support/zxdc-documentation) printable copies of all of Z/XDC's manuals. Each manual is available in PDF format.

Contents

PREFACE ii
PROPRIETARY LEGEND ii
CONTACTING COLESOFT ii
ONLINE PRESENCE iii
TRADEMARKS iii
PRODUCT MANUALS iii
ADDITIONAL MANUALS iv
CONTENTS v
INTRODUCTION 1
A Roadmap 1
ONLINE PRESENCE 2
Built-in Help Panels 5
List Help 9 5
Help 24
Help ABouthelp 26
Help ABouthelp Built-in 29
Help ABouthelp Online 32
Help ABouthelp PDfs 34
Help ABouthelp Conventions 36
Help ABouthelp PRinting 36
Help COmmands 37
Help CMddialogs 37
Help SHortcutcmds 38
Help SHortcutcmds ASSocaddr 39
Help SHortcutcmds ? 41
Help SHortcutcmds A 42
Help SHortcutcmds Cu 43
Help SHortcutcmds D 44
Help SHortcutcmds E 45
Help SHortcutcmds F 46
Help SHortcutcmds H 46
Help SHortcutcmds J 47
Help SHortcutcmds K 47
Help SHortcutcmds L 48
Help SHortcutcmds LL 50
Help SHortcutcmds LO 50
Help SHortcutcmds M 50
Help SHortcutcmds N 51
Help SHortcutcmds O 51
Help SHortcutcmds P 51
Help SHortcutcmds Q 52
Help SHortcutcmds S 52
Help SHortcutcmds SH 53
Help SHortcutcmds T 53
Help SHortcutcmds W 53
Help SHortcutcmds X 54
Help SHortcutcmds Z 55
Help SHortcutcmds ASTerisk 58
Help SHortcutcmds | 58
Help SHortcutcmds \ 58

Help SHortcutcmds / . . . 58
Help Pascmds . . . 59
Help Pascmds Syntax . . . 60
Help Pascmds Datafields . . . 62
Help Pascmds Instructions . . . 65
Help Pascmds Instructions F-example . . . 68
Help Pascmds Instructions T-example . . . 69
Help DEbugging . . . 73
Help DEbugging Gettingstarted . . . 74
Help DEbugging Amode64 . . . 77
Help DEbugging Batchjobs . . . 80
Help DEbugging C . . . 83
Help DEbugging C SEtup . . . 88
Help DEbugging C STarting . . . 89
Help DEbugging C Mapping . . . 93
Help DEbugging C Librarylists . . . 96
Help DEbugging C Librarylists Dwarf . . . 97
Help DEbugging C Librarylists Csource . . . 98
Help DEbugging C Librarylists Redirection . . . 99
Help DEbugging C Librarylists Lists . . . 100
Help DEbugging C Librarylists Lists REDirectlist . . . 101
Help DEbugging C Librarylists Lists REDirectlist Wildcards . . . 103
Help DEbugging C Librarylists Lists REDirectlist Matchvariables . . . 104
Help DEbugging C Librarylists Lists Candidateslists . . . 106
Help DEbugging C Librarylists Lists Candidateslists Resultobjects . . . 108
Help DEbugging C Librarylists Lists RESultslist . . . 110
Help DEbugging C Librarylists Lists Families . . . 111
Help DEbugging C Librarylists Lists Saving . . . 111
Help DEbugging C Librarylists Searchorder . . . 113
Help DEbugging C Librarylists Management . . . 114
Help DEbugging C Librarylists Glossary . . . 116
Help DEbugging C Debugging . . . 122
Help DEbugging C Variables . . . 124
Help DEbugging C CXDCHook . . . 126
Help DEbugging C CXDCIs . . . 127
Help DEbugging C Externalissues . . . 129
Help DEbugging CIcs . . . 130
Help DEbugging CIcs Xdccicsx . . . 134
Help DEbugging Dumps . . . 135
Help DEbugging EStaes . . . 135
Help DEbugging EStaes Removing . . . 137
Help DEbugging EStaes Superseding . . . 138
Help DEbugging EStaes Modifying . . . 140
Help DEbugging EStaes Modifying Informationalcalls . . . 145
Help DEbugging EStaes Modifying Example . . . 147
Help DEbugging EStaes Espies . . . 150
Help DEbugging EStaes Hooks . . . 151
Help DEbugging EStaes Debugging . . . 152
Help DEbugging EXitroutines . . . 153
Help DEbugging EXitroutines Examples . . . 155
Help DEbugging Frr . . . 161
Help DEbugging Hooks . . . 166
Help DEbugging Ispf . . . 166
Help DEbugging Ispf Panel . . . 169

Help DEbugging Ispf Panel Installing . . . 171
Help DEbugging Ispf DISplays . . . 172
Help DEbugging Ispf DIAlogs . . . 173
Help DEbugging Ispf Authorized . . . 174
Help DEbugging Ispf Xdclbdef . . . 176
Help DEbugging Jes2 . . . 178
Help DEbugging LE . . . 180
Help DEbugging LOstlocks . . . 182
Help DEbugging PCroutines . . . 184
Help DEbugging PLpa . . . 186
Help DEbugging REEntrant . . . 188
Help DEbugging REEntrant Storagekeys . . . 191
Help DEbugging REFreshable . . . 192
Help DEbugging REFreshable Storagekeys . . . 195
Help DEbugging RMode64 . . . 196
Help DEbugging Srbmode . . . 196
Help DEbugging TSo . . . 199
Help DEbugging Uss . . . 202
Help DEbugging Uss Operands . . . 203
Help DEbugging Uss Operands Uss= . . . 203
Help DEbugging Uss Operands Dub= . . . 204
Help DEbugging Uss Implementation . . . 205
Help DEbugging Uss Names . . . 206
Help DEbugging Uss Xdc . . . 210
Help DEbugging TIps . . . 210
Help DEbugging TIps Abendreascd . . . 211
Help DEbugging TIps Duplicates . . . 212
Help DEbugging TIps Duplicates Example . . . 215
Help DEbugging TIps S0c4 . . . 221
Help DEbugging TIps S0c4 Estaes . . . 221
Help DEbugging TIps S0c4 Badpointers . . . 223
Help DEbugging TIps S0c4 Initialregs . . . 223
Help DEbugging TIps S0c4 Systemcb . . . 223
Help ADdressing . . . 224
Help ADdressing Parsers . . . 228
Help ADdressing Parsers Parseorder . . . 228
Help ADdressing Parsers Tags . . . 229
Help ADdressing Parsers Asm . . . 230
Help ADdressing Parsers Asm MIxedcase . . . 232
Help ADdressing Parsers Asm Quoted . . . 234
Help ADdressing Parsers Asm RESolution . . . 236
Help ADdressing Parsers Asm RESolution Searches . . . 240
Help ADdressing Parsers Asm RESolution Examples . . . 241
Help ADdressing Parsers Asm DUplicates . . . 242
Help ADdressing Parsers Asm EXtents . . . 243
Help ADdressing Parsers Asm Longnames . . . 243
Help ADdressing Parsers Asm Asmequs . . . 244
Help ADdressing Parsers Asm Jpalpaplpa . . . 246
Help ADdressing Parsers Asm MOdules . . . 247
Help ADdressing Parsers Asm DSects . . . 248
Help ADdressing Parsers Asm EQuates . . . 249
Help ADdressing Parsers Asm REGisters . . . 249
Help ADdressing Parsers Asm Psws . . . 251
Help ADdressing Parsers Cee . . . 252

Help ADdressing Numeric . . . 253
Help ADdressing Numeric Scalingcodes . . . 255
Help ADdressing Dotplus . . . 256
Help ADdressing Implicit . . . 259
Help ADdressing Implicit Cdpndp . . . 260
Help ADdressing Targetspaces . . . 264
Help ADdressing Functions . . . 267
Help ADdressing Syntax . . . 268
Help ADdressing Syntax BAseterm . . . 269
Help ADdressing Syntax BAseterm Functions . . . 271
Help ADdressing Syntax BAseterm Registersandpsws . . . 272
Help ADdressing Syntax BAseterm Examples . . . 276
Help ADdressing Syntax Offsetterms . . . 281
Help ADdressing Syntax Offsetterms Constants . . . 282
Help ADdressing Syntax Offsetterms Operators . . . 283
Help ADdressing Syntax Offsetterms Functions . . . 283
Help ADdressing Syntax Offsetterms Examples . . . 285
Help ADdressing Syntax BEtweenterms . . . 287
Help ADdressing Syntax BEtweenterms Indirect . . . 287
Help ADdressing Syntax BEtweenterms Functions . . . 290
Help ADdressing Syntax BEtweenterms Examples . . . 290
Help ADdressing Examples . . . 292
Help FUNctions . . . 293
Help FUNctions CAtegories . . . 296
Help FUNctions CHart . . . 300
Help FUNctions RECursion . . . 301
Help FUNctions SYntax . . . 302
Help FUNctions FLc . . . 303
Help FUNctions ALet . . . 304
Help FUNctions ASId . . . 307
Help FUNctions ASN . . . 309
Help FUNctions F4 . . . 309
Help FUNctions H2 . . . 310
Help FUNctions ILC . . . 310
Help FUNctions IEq . . . 311
Help FUNctions ILIt . . . 312
Help FUNctions INdirect . . . 313
Help FUNctions Lmod . . . 317
Help FUNctions NXInst . . . 320
Help FUNctions NXSeq . . . 323
Help FUNctions Pclocate . . . 325
Help FUNctions RDdd . . . 326
Help FUNctions REAl . . . 328
Help FUNctions SAve4 . . . 330
Help FUNctions S1-thru-s8 . . . 330
Help FUNctions Width . . . 333
Help FUNctions XADDr . . . 334
Help FUNctions XADR . . . 336
Help FUNctions XALet . . . 336
Help FUNctions XASId . . . 338
Help FUNctions XASN . . . 340
Help FUNctions X1-thru-x8 . . . 340
Help FUNctions ZLeft . . . 343
Help FUNctions ZRight . . . 344

Help ATtentions . . . 345
Help ATtentions INXdc . . . 346
Help ATtentions INUserpgm . . . 346
Help ATtentions INThebatch . . . 349
Help Breakpoints . . . 349
Help Breakpoints Autocmds . . . 357
Help Breakpoints Conditional . . . 358
Help Breakpoints Conditional Watch . . . 363
Help Breakpoints DEAdtraps . . . 365
Help Breakpoints DEAdtraps #die . . . 366
Help Breakpoints DEFerred . . . 371
Help Breakpoints Families . . . 374
Help Breakpoints Persistent . . . 375
Help Breakpoints Reentrant . . . 375
Help Breakpoints Stepping-c . . . 376
Help Breakpoints Stepping-c C . . . 377
Help Breakpoints Stepping-c C Defaultstep . . . 378
Help Breakpoints Stepping-c C Pfkeys . . . 378
Help Breakpoints Stepping-c C MIxwithtrace . . . 379
Help Breakpoints Stepping-c C MAChinecode . . . 379
Help Breakpoints Stepping-c C Autostep . . . 380
Help Breakpoints Stepping-c C MAPsrequired . . . 380
Help Breakpoints TRANsient . . . 381
Help Breakpoints TRAPping . . . 383
Help Breakpoints TRACing . . . 385
Help Breakpoints TRACing Branches . . . 389
Help Breakpoints TRACing STorealter . . . 393
Help Breakpoints TRACing SUbroutines . . . 396
Help Breakpoints TRACing Alien . . . 398
Help Breakpoints TRACing Ex/exrl . . . 401
Help Breakpoints TRACing Pfkeys . . . 402
Help Breakpoints TRACing Conditional . . . 406
Help Breakpoints TRACing Conditional Multiple . . . 410
Help Breakpoints TRACing Conditional Corruptions . . . 412
Help Breakpoints TRACing Conditional Performance . . . 416
Help Breakpoints TRACing Conditional Slipvsxdc . . . 418
Help Breakpoints TRACing Conditional Failures . . . 420
Help Breakpoints TYpes . . . 423
Help Breakpoints TYpes Corruption . . . 428
Help CHaractersets . . . 429
Help Hooks . . . 431
Help Hooks DYnamic . . . 433
Help Hooks DYnamic Security . . . 435
Help Hooks DYnamic Homespace . . . 436
Help Hooks DYnamic Otherspaces . . . 438
Help Hooks DYnamic Otherspaces Startingnewsessions . . . 439
Help Hooks DYnamic Commonstorage . . . 440
Help Hooks DYnamic Whathappens . . . 442
Help Hooks Static . . . 445
Help Hooks Static #xdchook . . . 446
Help Hooks Static Cxdchook . . . 453
Help Hooks DEferred . . . 453
Help Hooks Legacysupport . . . 455
Help Hooks Undoing . . . 455

Help DDnames . . . 456
Help DDnames ADAta . . . 459
Help DDnames ADUmp . . . 460
Help DDnames CDF . . . 460
Help DDnames CDFTrace . . . 460
Help DDnames CMds . . . 460
Help DDnames COff . . . 461
Help DDnames Fptnn . . . 461
Help DDnames Itrac . . . 462
Help DDnames JSTCb . . . 462
Help DDnames JSTEp . . . 462
Help DDnames Lib . . . 462
Help DDnames Maplb . . . 463
Help DDnames NComp . . . 463
Help DDnames NOAlc . . . 463
Help DDnames NOMap . . . 464
Help DDnames NOSc . . . 464
Help DDnames NStep . . . 464
Help DDnames PLib . . . 464
Help DDnames PRInt . . . 465
Help DDnames PROf . . . 465
Help DDnames Quick . . . 465
Help DDnames REFr8 . . . 466
Help DDnames RENt8 . . . 466
Help DDnames REXx . . . 467
Help DDnames SDump . . . 467
Help DDnames XClud . . . 468
Help DDnames SYSAdata . . . 469
Help DDnames SYSCdbg . . . 470
Help DDnames SYSMdump . . . 470
Help DDnames SYS000nn . . . 470
Help DDnames TAsklib . . . 471
Help DDnames TLib . . . 471
Help DDnames TRAce . . . 472
Help DDnames TRSaf . . . 472
Help DDnames TSprt . . . 472
Help DDnames Wtor . . . 473
Help DDnames XDcis . . . 473
Help DUmps . . . 474
Help DUmps Startingdumpxdc . . . 476
Help DUmps Startingdumpxdc Dxdcproc . . . 480
Help DUmps Startingdumpxdc Dxdcproc Arguments . . . 482
Help DUmps Startingdumpxdc Processingoptions . . . 483
Help DUmps Startingdumpxdc Processingoptions Asdefault . . . 489
Help DUmps Startingdumpxdc Processingoptions Mmef . . . 490
Help DUmps Startingdumpxdc Processingoptions Summarydata . . . 490
Help DUmps Startingdumpxdc Switching . . . 490
Help DUmps Startingdumpxdc Tsoready . . . 491
Help DUmps Startingdumpxdc Batch . . . 493
Help DUmps DUmptypes . . . 496
Help DUmps DUmptypes Slip . . . 498
Help DUmps CONtents . . . 499
Help DUmps CONtents Foreignvlocal . . . 502
Help DUmps DIsagreementswithipcs . . . 504

Help DUmps DIsagreementswithipcs Pswregs . . . 505
Help DUmps EXPecteduses . . . 505
Help DUmps EXPecteduses Personaluse . . . 507
Help DUmps EXPecteduses Personaluse Sysmdumps . . . 508
Help DUmps EXPecteduses Personaluse C . . . 510
Help DUmps EXPecteduses Personaluse Asm . . . 512
Help DUmps EXPecteduses Isvusage . . . 514
Help DUmps MAppingdata . . . 515
Help DUmps MAppingdata Bindermaps . . . 519
Help DUmps MAppingdata Adatamaps . . . 522
Help DUmps MAppingdata Symdatamaps . . . 525
Help DUmps MAppingdata DWarfmaps . . . 527
Help DUmps MAppingdata Csource . . . 528
Help DUmps MAppingdata Literals . . . 529
Help DUmps MAppingdata Mmeffiles . . . 532
Help DUmps MAppingdata DEfine . . . 534
Help DUmps MAppingdata Journaling . . . 541
Help DUmps MAppingdata Zoscblocks . . . 542
Help DUmps Literals . . . 544
Help DUmps Literals Dump/xdc . . . 544
Help DUmps Literals Ipcs . . . 548
Help DUmps EXEcutionthread . . . 548
Help DUmps EXEcutionthread Initial . . . 552
Help DUmps EXEcutionthread Changing . . . 557
Help DUmps EXEcutionthread Errorandretrylevels . . . 564
Help DUmps EXEcutionthread Srbmode . . . 566
Help DUmps COMmands . . . 567
Help DUmps MIscellaneous . . . 567
Help DUmps MIscellaneous DDnames . . . 567
Help DUmps MIscellaneous Locks . . . 568
Help DUmps MIscellaneous Profiles . . . 569
Help DUmps MIscellaneous Closed . . . 571
Help DUmps MIscellaneous Redacted . . . 571
Help DUmps MIscellaneous DAtaspaces . . . 572
Help EQuates . . . 572
Help EQuates Builtin . . . 575
Help EQuates Builtin Cblocks . . . 576
Help EQuates Builtin Storage . . . 577
Help EQuates Builtin REGSets . . . 579
Help EQuates Builtin REGIsters . . . 580
Help EQuates Builtin Internal . . . 582
Help EQuates Builtin Other . . . 583
Help EQuates Builtin Automatic . . . 585
Help EQuates Floating . . . 590
Help EQuates Autocloning . . . 593
Help EQuates Ritargets . . . 594
Help EXEcutionlevels . . . 594
Help EXEcutionlevels RBs . . . 600
Help EXEcutionlevels RBs Resumeaddress . . . 604
Help EXEcutionlevels Errorlvl . . . 606
Help EXEcutionlevels REtrylvl . . . 608
Help EXEcutionlevels REtrylvl Types . . . 609
Help EXEcutionlevels REtrylvl ESTAEs . . . 611
Help EXEcutionlevels REtrylvl ESTAIs . . . 612

Help EXIts . . . 614
Help EXIts Acif . . . 615
Help EXIts Frrend . . . 617
Help EXIts Init . . . 619
Help EXIts Miscxits . . . 619
Help EXIts RESume . . . 622
Help EXIts Termin8 . . . 623
Help EXIts UCi . . . 623
Help EXIts USErcmds . . . 623
Help EXIts USErcmds Srcxucmd . . . 626
Help EXIts USVctrce . . . 627
Help EXIts REGstack . . . 627
Help EXIts REGstack Srcxureg . . . 631
Help FULlscreen . . . 632
Help FULlscreen TErminals . . . 633
Help FULlscreen TErminals Geometries . . . 633
Help FULlscreen TErminals Ispfsettings . . . 638
Help FULlscreen TErminals Colors . . . 638
Help FULlscreen TErminals Bindimages . . . 639
Help FULlscreen TErminals Problems . . . 640
Help FULlscreen TItle . . . 641
Help FULlscreen TItle Center . . . 642
Help FULlscreen TItle Left . . . 644
Help FULlscreen TItle Right . . . 645
Help FULlscreen Fsinputs . . . 647
Help FULlscreen Displayerrs . . . 648
Help FULlscreen Ispf . . . 649
Help FULlscreen Logging . . . 650
Help FULlscreen Logging Latentcommands . . . 652
Help FULlscreen Notes . . . 653
Help FULlscreen PFkeys . . . 653
Help FULlscreen PFkeys HYphen . . . 654
Help FULlscreen PFkeys Underscore . . . 655
Help FULlscreen PFkeys Pfkeysets . . . 656
Help FULlscreen PFkeys Pfkeysets Suggestions . . . 657
Help FULlscreen PFkeys Defaultkeys . . . 659
Help FULlscreen PFkeys Defaultkeys SET-A . . . 660
Help FULlscreen PFkeys Defaultkeys SET-B . . . 664
Help FULlscreen PFkeys HElpkeys . . . 667
Help FULlscreen Retrieve . . . 670
Help FULlscreen SCrolling . . . 672
Help FULlscreen SCrolling Storagescrolling . . . 676
Help FULlscreen Windows . . . 678
Help FULlscreen Windows Primarywindow . . . 679
Help FULlscreen Windows WAtchwindows . . . 680
Help MACros . . . 681
Help MACros #Cblocks . . . 682
Help MACros #DBCParm . . . 682
Help MACros #DBCR0In . . . 688
Help MACros #DBCR0Ut . . . 689
Help MACros #DBCVrsn . . . 690
Help MACros #DIe . . . 690
Help MACros @symoff/on . . . 690
Help MACros #Test . . . 693

Help MACros #XDCHook . . . 694
Help MACros #XDCLoc8 . . . 694
Help MAPs . . . 695
Help MAPs AUTOMapping . . . 702
Help MAPs Bindermaps . . . 702
Help MAPs CSECTMaps . . . 705
Help MAPs CSECTMaps Asmoffsets . . . 707
Help MAPs DSectmaps . . . 709
Help MAPs CMaps . . . 710
Help MAPs CSECTSasdsects . . . 711
Help MAPs Privatelyloaded . . . 715
Help MAPs Privatelyloaded Dmapandusing . . . 716
Help MAPs Privatelyloaded Map . . . 718
Help MAPs Privatelyloaded Ussfiles . . . 720
Help MAPs Sym . . . 721
Help MAPs Sym Excesssymdata . . . 722
Help MAPs Sym Excesssymdata Examples . . . 724
Help MAPs Sym Excesssymdata Xdcsymed . . . 726
Help MAPs Sym Excesssymdata Tachyon . . . 728
Help MAPs AData . . . 729
Help MAPs AData CReating . . . 731
Help MAPs AData MAPlibs . . . 732
Help MAPs AData MAPlibs Namedlists . . . 736
Help MAPs AData Searchorder . . . 737
Help MAPs AData CAching . . . 738
Help MAPs AData Performance . . . 739
Help MAPs AData Excessadata . . . 740
Help MAPs AData Excessadata Examples . . . 744
Help MAPs AData Excessadata Xdcadata . . . 745
Help MAPs AData MACros . . . 752
Help MAPs AData Tips . . . 753
Help MAPs DWarf . . . 753
Help MAPs Floating . . . 754
Help MAPs AUTOCloning . . . 757
Help MAPs Xdcmaps . . . 758
Help MAPs Xdcmaps Adata . . . 762
Help MAPs Xdcmaps Yourdoc . . . 764
Help MAPs CIcsmaps . . . 764
Help MEssages . . . 766
Help MUltitask . . . 766
Help MUltitask Scope . . . 767
Help MUltitask Conversationmanagement . . . 770
Help MUltitask Irbs . . . 771
Help PRofiles . . . 773
Help PRofiles MEnu . . . 773
Help PRofiles MEnu TSoispfcdfcics . . . 776
Help PRofiles MEnu TErmsettings . . . 778
Help PRofiles MEnu COlors . . . 780
Help PRofiles MEnu Pfktofkeymap . . . 783
Help PRofiles MEnu Keys . . . 784
Help PRofiles MEnu Hkeys . . . 786
Help PRofiles MEnu Log . . . 787
Help PRofiles MEnu Windows . . . 791
Help PRofiles MEnu REAdzapparse . . . 792

Help PRofiles MEnu REAdzapparse SToreprotzap . . . 793
Help PRofiles MEnu REAdzapparse Mnemoniczap . . . 794
Help PRofiles MEnu REAdzapparse Readecho . . . 795
Help PRofiles MEnu REAdzapparse SEqfield . . . 795
Help PRofiles MEnu REAdzapparse SCriptlibname . . . 796
Help PRofiles MEnu REAdzapparse Parseorder . . . 797
Help PRofiles MEnu Formattracefind . . . 798
Help PRofiles MEnu Formattracefind SOurceformatting . . . 799
Help PRofiles MEnu Formattracefind Macroexpansions . . . 800
Help PRofiles MEnu Formattracefind FOrmattingbias . . . 801
Help PRofiles MEnu Formattracefind SToragelocations . . . 801
Help PRofiles MEnu Formattracefind DISPLACementfields . . . 802
Help PRofiles MEnu Formattracefind TExtencoding . . . 803
Help PRofiles MEnu Formattracefind DISPLAYwidths . . . 803
Help PRofiles MEnu Formattracefind Pointerresolutions . . . 804
Help PRofiles MEnu Formattracefind TRACEPositioning . . . 805
Help PRofiles MEnu Formattracefind TRACEBasr . . . 806
Help PRofiles MEnu Formattracefind Breakpointopcode . . . 807
Help PRofiles MEnu Formattracefind DEadtrapbypass . . . 808
Help PRofiles MEnu Formattracefind FIndformatting . . . 808
Help PRofiles MEnu Automapprintetc . . . 809
Help PRofiles MEnu Automapprintetc Automap . . . 809
Help PRofiles MEnu Automapprintetc Lpp . . . 810
Help PRofiles MEnu Automapprintetc SYsoutclass . . . 810
Help PRofiles MEnu Automapprintetc Holdnohold . . . 810
Help PRofiles MEnu Automapprintetc CAnceldetachdebugging . . . 811
Help PRofiles MEnu Automapprintetc SEtbang . . . 811
Help PRofiles MEnu Automapprintetc COnversationtimeout . . . 812
Help PRofiles MEnu Automapprintetc Flc . . . 813
Help PRofiles MEnu REXx . . . 813
Help PRofiles MEnu CXdcsettings . . . 814
Help PRofiles MEnu Uss . . . 814
Help PRofiles MApliblists . . . 815
Help PRofiles Readsave . . . 816
Help PRofiles Namedprofiles . . . 817
Help PRofiles Namedprofiles Example . . . 819
Help PRofiles Profnames . . . 819
Help PRofiles DEfaultprofile . . . 820
Help PRofiles Factorydefaults . . . 823
Help PRofiles Factorydefaults A80 . . . 824
Help PRofiles Factorydefaults AWide . . . 826
Help PRofiles Factorydefaults C80 . . . 828
Help PRofiles Factorydefaults CWide . . . 830
Help PRofiles Oldprofiles . . . 832
Help PRofiles DDnames . . . 832
Help PRofiles Tlibprofiles . . . 833
Help PRofiles Wide . . . 834
Help Rexx . . . 835
Help Rexx Environment . . . 836
Help Rexx SAYoutput . . . 837
Help Rexx Input . . . 838
Help Rexx XDCServices . . . 839
Help Rexx XDCServices XDCCmd . . . 841
Help Rexx XDCServices XDCDump . . . 842

Help Rexx XDCServices XDCGet . . . 843
Help Rexx XDCServices XDCMemry . . . 843
Help Rexx XDCServices XDCParse . . . 845
Help Rexx XDCServices XDCRDisp . . . 847
Help Rexx XDCServices XDCROute . . . 847
Help Rexx XDCServices XDCSAy . . . 849
Help Rexx XDCServices XDCSTate . . . 850
Help Rexx XDCServices XDCVars . . . 854
Help Rexx XDCServices XDCZAp . . . 855
Help Rexx XDCServices XDCZStat . . . 857
Help Rexx Userfunctions . . . 859
Help Rexx XDCCommands . . . 860
Help Rexx SAMpleexecs . . . 860
Help SEcurity . . . 862
Help SEcurity Rules . . . 865
Help SEcurity AUth . . . 866
Help SEcurity CDf . . . 867
Help SEcurity Fasm . . . 869
Help SEcurity Lostlocks . . . 871
Help SEcurity Zap . . . 873
Help SEcurity Zap Description . . . 874
Help SEcurity Zap Oldrules . . . 877
Help SEcurity Zap Suggestions . . . 879
Help SEcurity Gzap . . . 881
Help SEcurity ACif . . . 881
Help SEcurity CAche . . . 882
Help SEcurity Trace . . . 882
Help Tachyon . . . 883
Help USERInterfaces . . . 885
Help USERInterfaces Fullscreen . . . 886
Help USERInterfaces Cdf . . . 888
Help USERInterfaces Linemode . . . 889
Help USERInterfaces Wtowtor . . . 889
Help USERInterfaces Uci . . . 890
Help UTilities . . . 891
Help UTilities XDCAdata . . . 891
Help UTilities XDCCall . . . 892
Help UTilities XDCMmef . . . 893
Help UTilities XDCSymed . . . 895
Help Virtmem . . . 895
Help Virtmem Concepts . . . 896
Help Virtmem ACcesslists . . . 896
Help Virtmem ALets . . . 897
Help Virtmem Xdcaccess . . . 898
Help Virtmem Xdcaccess Fasm . . . 899
Help Virtmem Xdcaccess Fasm Real . . . 901
Help Virtmem Xdcaccess Lasm . . . 902
Help Virtmem Mvsdoc . . . 903
Help XDCCAll . . . 903
Help XDCCAll ENvironment . . . 909
Help XDCCAll Interferingestaes . . . 911
Help XDCCAll EXits . . . 912
Help XDCCAll Tasklib . . . 913
Help XDCCAll Tasklib Tasklibraries . . . 915

Help XDCCAll Jobsteptasks . . . 915
Help XDCCAll Ddnames . . . 916
Help XDCCLones . . . 918
Help XDCCLones Using . . . 919
Help XDCCLones Axdcis . . . 922
Help XDCCLones Cxdcis . . . 923
Help XDCCLones Profiles . . . 925
Help XDCSrver . . . 926
Help XDCSrver Isbouncingsafe? . . . 928
Help XDCSrver Cdf . . . 931
Help XDCSrver Cdf Setup . . . 933
Help XDCSrver Cdf COMMUnications . . . 937
Help XDCSrver Cdf CONnection . . . 938
Help XDCSrver Cdf Tso . . . 940
Help XDCSrver Cdf Logon . . . 941
Help XDCSrver Cdf COMMAnds . . . 943
Help XDCSrver Cdf Fasm . . . 944
Help XDCSrver Cdf Recovery . . . 947
Help XDCSrver Cdf CRosssysplex . . . 949
Help SUpport . . . 950
Help SUpport Featuresandcaps . . . 951
Help SUpport Hlq . . . 954
Help SUpport Manuals . . . 954
Help SUpport Online . . . 955
Help SUpport Online Website . . . 955
Help SUpport Online Facebook . . . 957
Help SUpport Online Youtube . . . 957
Help SUpport COntactus . . . 957
Help SUpport SEndingusdumps . . . 958
Help SUpport SUpportstatement . . . 960
Help SUpport Legalnotice . . . 961
Help SUpport CRedits . . . 962
Help Whatsnew . . . 965

INTRODUCTION

ColeSoft has pursued the goal of making Z/XDC's internal documentation as comprehensive as possible. Towards that end, we have devoted considerable effort to greatly expanding the amount of information available within Z/XDC and to improving the accessibility of that information and the navigability of the Help Database as a whole.

This manual is nothing more than a printout of a section of the Help Database. It is provided for those people (like myself) who steadfastly prefer looking at paper instead of glass. (It's hard to write margin notes on glass.)

The information in the Help Database has been segmented into four printed documents:

- Z/XDC® User Guide
  Contains comprehensive tutorials about the many features and capabilities of Z/XDC.
- Z/XDC® Commands
  Contains the detailed syntax, usage descriptions, and examples of all of Z/XDC's commands.
- Z/XDC® Messages
  Contains descriptions of all of the messages that can be issued by Z/XDC and all of its various components.
- Z/XDC® V3.1 Release Guide
  Contains a history of all changes and upgrades made in the current release of Z/XDC.

There are a couple of important structural differences between Z/XDC's internal Help and these manuals:

- The PDF copies of the printed manuals can be searched using typical PC-style searching commands.
- "Release Guides" for older versions and releases of Z/XDC are available only via the "HELP WHATSNEW" command.

A Roadmap

The structure of this manual follows the structure of the Help Database. A consequence of this is that the sequence of information in this book, over all, is decidedly non-sequential. For those of you who prefer to read a manual from beginning to end, please accept my apologies. However, please let me make some suggestions.

If you are an experienced Z/XDC user, then start with the **Z/XDC® V3.1 Release Guide**. This will tell you what's new in this release of Z/XDC. Within Z/XDC, the Release Guide can be reached by typing HELP WHATSNEW. You can then use hyperlinks to pursue the specific information that is of interest to you.

For new users, turn to the **Z/XDC® User Guide**, and examine its Table of Contents carefully. You will see that there are about two dozen major topics arranged alphabetically: Addressing, Attentions, Breakpoints, ..., Virtmem, XDCCALL. Information within topics is presented more or less sequentially. The following **User Guide** topics are of particular interest:

- Perhaps the first topic that should be read is named **"DEBUGGING"**. This and its subtopics give comprehensive information about whether and to what extent you may have to modify your program in order to use Z/XDC.

- Another topic that should be read early on is named **"XDCCALL"**. XDCCALL is a utility program that can be used to start a debugging session for your program.
- If you plan to debug programs that run as batch jobs or system tasks, then read the **"CDF"** topic. "Cross Domain Facility" is the component of Z/XDC that permits user terminals to connect to debugging sessions for background jobs.

For Z/XDC command information, turn to the **Z/XDC® Commands** manual. Start with the basic commands. The DISPLAY, FORMAT, and LIST commands display storage and important program related structures. The AT and TRAP commands set breakpoints. You can use the TRACE command to step execution through your program slowly. The ZAP command allows you to change storage and registers.

If you wish to play with Z/XDC's terminal and user interfaces, read the **"FULLSCREEN"** section of the **User Guide**. Also, try the PROFILE command for displaying and changing a very large number of session parameters.

Generally, the best approach is to plan your reading using the Table of Contents. And of course, if you can't find the information that you are looking for, call us. There's no charge, and we will be glad to help! Our number is 800-XDC-5150 (USA: 928-771-2003). If the information that you want is in the book, we will explain what you want to know and tell you where to find complete information. If it is not, then we will add it for our next release.

ONLINE PRESENCE

ColeSoft Marketing maintains the following resources on the Internet:

[Home Page] ColeSoft's Home Page is www.colesoft.com. It provides the following services:

- General information about Z/XDC
- E-mail links to both Marketing, Technical Support, and Customer Services
- FTP links for uploading diagnostic information and other files to Technical Support
- A dialog for downloading current maintenance for Z/XDC
- Links permitting existing customers to download a full set of Z/XDC's documentation
- Online product delivery
- 24x7 self-service for temporary, short-term, license activation codes for use in D.R. tests and other emergencies

[Facebook] ColeSoft's Facebook presence is at facebook.com/colesoftware. This is where we will from time to time post information about ColeSoft people and activities.

[LinkedIn] ColeSoft has a users group named Z/XDC Users Group. This is the "Go-To" place for all things Z/XDC. So if you want to see what's coming up with Z/XDC, then join this group. Things that we put here include:

- Notices about new releases and what they include
- Notices of new maintenance and what has been fixed, changed or added
- Notices of new training videos as we create them
- Creative ways to solve situations that our customers might encounter
- Short "how to" tips illustrating now to use Z/XDC and what it can do

But we want this group to be a two-way street. We would love it if our customers would post to the group such things as:

- Questions about how to do something with Z/XDC
- Suggestions about how to improve Z/XDC
- Interesting experiences customers have had using Z/XDC
- New ways to use Z/XDC that make you smile
- Problems encountered with Z/XDC that you would like help with
- Pretty much anything having to do with Z/XDC

Built-in Help Panels

List Help 9

```
Help                          ...2503A-rp-not-correct       ...2401A-Product-library-hl
.ABouthelp                    ...2502E-xdcadata-changes     ...2401A-VTam-emsging
..Built-in                    ...2502D-server-cdf-s0c4      ...2401A-VCeqbs-and-vceqh-s
..Online                      ...2502C-passphrase-with-sp   ..2023
..PDfs                        ...2502A-rexx-changes         ...2312b-missing-dfp-instru
..Conventions                 ..2024                        ...2311C-set-format-decimal
..PRinting                    ...2412a-dead-trap-9428       ...2311B-dead-trap-#3488
.Maintenance                  ...2411C-XDCLBDEF-Changes     ...2311A-s72a-abend-in-cdf
..2026                        ...2411C-XDCLBDEF-Help        ...2310a-list-ls
...2609B-modify-operator-co   ...2411A-csample-name-colli   ...2309E-install-improvemen
...2609A-intermittent-500-h   ...2410a-storage-pointers-r   ...2309D-adata-exclusion-li
...2608D-iefu86-support       ...2409I-c/xdc-variable-dis   ...2309C-c/xdc-hang
...2608C-dbc502-error         ...2409H-list-searchorder     ...2309B-adata-changes
....2608B-v3 1-doc-fixes      ...2409E-c/xdc-variable-dis   ...2309A-srb-bundle-corrupt
...2608A-batch-tso-ispf-rc    ...2409C-dbc884i              ...2308J-Install-failed-wit
...2607E-vscode-extension     ...2409B-MEssage-clawback-a   ...2308J-New-doc-for-sn-and
...2607D-list-psw-not-repor   ...2409B-MInor-changes        ...2308I-map-xlpa
...2607C-list-nt-command      ...2409A-minor-fixes          ...2308H-user-alet
...2607B-list-feature-iaf2    ...2408B-Profile-menuing-sy   ...2308F-c/xdc-list-cxdc-ca
...2607A-list-asid-wildcard   ...2408B-Misc-minor-changes   ...2308E-dbc350-maint
...2606a-load-failure-in-sr   ...2408A-c/xdc-variable-dis   ...2308D-list-variables-s0c
...2604B-profile-read-error   ...2407B2-dump/xdc-msg-help   ...2308B-customer-hlq
...2604A-fix-dead-8360        ...2407B1-find-command        ...2308A-0e0-2d-abends
...2603D-reduce-xdc-size      ...2407A-Call-interface-cha   ...2307C-adata-0c4
...2603C-load-failure-pem-2   ...2407A-#dbcparm-macro-cha   ...2307B-list-rbs
...2603B-t-bnc-pc-instructi   ...2407A-New-doc-for-basr-d   ...2306F-cross-memory-issue
...2603A-list-asid-messages   ...2406C-missing-messages     ....H2306b-cics-6 1-support
...2602a-dump/xdc-error       ...2406B-abend0c4-list-vari   ...2306A-c/xdc-fixes-and-en
...2601B-mdat-debug-code      ...2406A-go-releasecaps-bug   ...2305D-dump/xdc-no-wtor-c
...2601A-uss-debugging-2      ...2405E-pem-2405a-in-error   ...2305C-fix-error
..2025                        ...2405D-bug#2-in-2405b       ...2305A-losing-amode-from-
...2511C-pem-2306f-in-error   ...2405C-bug-in-2405b         ...2304J-cdf/xdc-recurisve-
...2511B-list-pc#-enh         ...2405B-minor-changes        ...2304I-cdf/xdc-error-reco
...2511A-zap-protected-mobs   ...2405A-pem-2403b-in-error   ...2304G-Ipcs-batch-abends
...2507E-I-changed-to-ll      ...2404D-pem-2404a-in-error   ...2304G-Command-help-dump/
...2507E-List-locations-twe   ...2404C-c/xdc-caps-server-   ...2304G-Literal-help-dump/
...2507D-dump/xdc-registers   ...2404A-list-rsa             ...2304G-Slip-dump-messages
...2507C-c/xdc-list-variabl   ...2403G-test-update          ...2304F-find-command-typo
...2507B-list-ispf-not-find   ...2403F-help-abouthelp       ...2304E-new-help-topic
...2506D-list-xdc-output-br   ...2403E-xdcadata-s0c4        ...2304D-fixed-list-searcho
...2506B-server/xdc-estae     ...2403D-rmctz-added-to-xdc   ...2304C-summary-dump-messa
...2506A-z17-opcodes          ...2403C-help-online          ...2304B-dead-trap-9654
...2505b-24-bit-constraint-   ...2403B-uss-debugging        ...2304A-profile-save-empty
...2504D-dbc539e-added        ...2403A-show-dbc040e-dsect   ...2303C-dump/xdc-interrupt
...2504C-find-interval        ...2402E-disconnect-abends-   ...2303B-dump/xdc-list-dspa
...2504B-flc-uss              ...2402C-cdf-router-task-se   ...2303A-FIXED-LIST-STglayo
...2503D-Disasm-regs-refs     ...2402B-c/xdc-wsa-fetchep    ...2303A-FIXED-LIST-SEarcho
...2503D-Rwn-changed-to-grn   ...2402A-stop522-failed       ...2302K-dump/xdc-title-cor
...2503D-Xdcadata-doc         ...2401A-Bouncing-server-is   ...2302H-cwide-performance-
```

...2302G-FIXED-LIST-Rbs
...2302G-FIXED-LIST-Lstack
...2302F-list-rsa-output
...2302E-new-commands
...2302D-rexx-changes
...2302C-proxy-task-search-
...2302B-cxdchook64-s0c1
...2301H-HELP-TOPIC-Renamed
...2301H-HELP-TOPIC-Source-
...2301H-Set-and-list-panel
...2301F-lcd-mgmt-added-to-
...2301E-c/xdc-f6sa-abend0c
...2301D-Fixed-commands
...2301D-Options-changed-fo
...2301D-Cxu-changed-for-du
...2301C-format-equ-star-mi
...2301B-customer-assembly-
...2301A-#xdchook-avoiding-
..2022
...2212K-maintenance-proces
...2212J-zacs-compliance
...2212I-fixed-server-work-
...2212G-list-fregs
...2212F-maintenance-proces
...2212D-list-rsa
...2212C-c/xdc-map-xplink-p
...2212B-register-equates
...2212A-changes-to-find-an
...2211B-Fixed-0c4-in-list-
...2211B-Doc-post-command-f
...2211A-fixed-trace-crt-an
...2210D-no-dbc790-when-tra
...2210C-c/xdc-list-vstack
...2210B-pr-trap2
...2210A-c/xdc-fam-array-su
...2209D-prep-for-upcoming-
...2209B-list-enq-fixes
...2209A-cross-memory-retry
...2208a-delete-modules-mes
...2207D-performance-bug-fl
...2207C-equates-added-out-
...2207A-set-asid-and-list-
...2206D-z16-machine-instru
...2206C-T-Bnc
...2206C-Autotbnc
...2206C-SVctable
...2206C-T-Sb/sa
...2206C-Estae-detection
...2206C-LIST-BREakpoints
...2206C-LIST-BRAnches
...2206C-Off-command-issues
...2206C-S-q-silent=yes
...2206C-Doc-changes
...2206B-dump/xdc-loop
...2206A-xdclcnse-is-not-mo
...2205H-list-lstack
...2205G-wrong-offsets
...2205F-c/xdc-list-variabl
...2205E-map-command-ddname
...2205D-list-location-comm
...2205C-cxu-problems
...2205B-dxdc-procedure-arg
...2205A-dump/xdc-ipcs-diff
...2204G-list-ssrbs-under-d
...2204F-c/xdc-list-variabl
...2204E-mapping-abend
...2204D-dump/xdc-fixes
...2204C-bea
...2204A-dump/xdc-fixes
...2203B-c/xdc-xlclang
...2203A-dump/xdc-fixes
...2202F-dump/xdc-ipcs-rout
...2202E-dbc388-message
...2202D-@tea-bi-always-exi
...2202C-fix-for-mdl-2201v
...2202B-dump/xdc-terminal-
...2202A-freemain-failures
...2201V-c/xdc-source-page-
...2201T-show-h-shortcut
...2201S-list-enq-under-dum
...2201R-dump/xdc-list-dspa
...2201Q-doc-changes
...2201O-waiting-rb-indicat
...2201N-sh-shortcut
...2201M-maintenance-level-
...2201K-dump/xdc-list-dspa
...2201J-list-dxdc
...2201I-list-set-zap
...2201H-incident-token-for
...2201G-list-variables-sho
...2201F-builtin-equate-nam
...2201E-dump/xdc-playback-
...2201D-dump/xdc-ipcs-vx-h
...2201C-dbc159e-dfe-diagno
...2201B-new-map-option-1
...2201A-dmap-uselds-option
..2021
...2112S-dbc159e-and-autocl
...2112R-list-subpools-impr
....2112Q-cics5 6-support
...2112P-error-in-pem-2112j
...2112O-playback-response
...2112N-xdcmmef-extract-dl
...2112M-dump/xdc-loop-aben
...2112L-c/xdc-dbc315e
...2112K-mmef-vers-tol-2
...2112J-dump/xdc-extractio
...2112I-c/xdc-dw-op-ibm-lo
...2112H-buf-oflow-in-list-
...2112G-dead-#1611-when-ie
...2112F-dump/xdc-s202-shut
...2112E-dump/xdc-dead-trap
...2112D-dump/xdc-bad-extra
...2112C-dump/xdc-handling-
...2112B-map-dlpa
...2112A-added-ipcs-equate-
...2111G-misc-minor-updates
...2111F-c/xdc-list-vars-md
...2111E-new-command-list-p
...2111D-c/xdc-dump/xdc-met
...2111C-mmef-vers-tol
...2111B-new-option
...2111A-reserved-renamed-t
...2110I-help-4digit-msg-nu
...2110H-dump/xdc-not-journ
...2110G-c/xdc-map-metalc
...2110F-dump/xdc-ipcs-comm
...2110E-dump/xdc-srb-mode-
...2110D-macro-opsyn-anop-i
...2110C-dump/xdc-bad-oper-
...2110B-dump/xdc-expand-is
...2110A-c/xdc-map
...2109K-fixed-cdf-connect-
...2109J-nucleus-map-takes-
...2109I-format-command-fai
...2109H-c/xdc-map-failure-
...2109F-dump/xdc-not-handl
...2109E-dump/xdc-not-findi
...2109D-dead-7166-in-forma
...2109C-c/xdc-map-dwarf-no
...2109B-added-ipcs-literal
...2109A-fixed-command-pars
...2108O-fixed-asm-err-in-#
...2108N-fixed-dump-xdc-bat
...2108M-diag-summary-dump-
...2108L-fixed-dead-trap-42
...2108K-fixed-cdf-connect-
...2108I-doc-maint-help-reo
...2108G-fixed-list-psw-dum
...2108F-Fixed-dead-trap-72
...2108F-Doc-changes
...2108E-tweak-end-and-list
...2108D-new-dump/xdc
...2108C-doc-maintenance
...2108B-improved-cda-mappi
...2108A-New-t-pascmd
...2108A-Doc-h-pascmd-rewri
...2107I-fixed-c-character-
...2107F-CHANGED-@rwn-exist
...2107F-CHANGED-Rhn-in-use
...2107F-CHANGED-Deleting-f
...2107F-FIXED-Freemain-equ

...2107F-FIXED-Equate-priva
...2107C-fixed-xdcsrver-@$g
...2107B-fixed-trace-crt-an
...2107A-Offsets-shown-in-a
...2107A-FIXED-PAs-failing-
...2107A-IMPROVED-@symoff/o
...2107A-Changed-dc/ds-when
...2107A-FIXED-Currentmcode
...2107A-FIXED-POsitioning-
...2107A-IMPROVED-New-windo
...2106E-FIXED-Dsect-prefix
...2106E-CHANGED-Bare-dot
...2106E-CHANGED-Dot-plus-o
...2106E-FIXED-Offsets-in-a
...2106D-coping-bad-dwarf-f
...2106C-diag-list-vars-0c4
...2106B-fixed-trace-sb
...2106A-changed-format-lin
...2105C-IMPROVED-Multiple-
...2105C-FIXED-Hidemcode-c8
...2105C-FIXED-Maxlines-ren
...2105C-IMPROVED-Scroll-ar
...2105B-new-cxdctask-secur
...2104C-improved-dwarf-dsn
...2104B-IMPROVED-Topic-pos
...2104B-Changed-showmcode-
...2104B-IMPROVED-Scanlog-n
...2104B-New-s-and-h-shortc
...2104A-new-xdccicsm
...2103J-fixed-dwarf-mappin
...2103I-CHANGED-Locate-nnn
...2103I-CHANGED-Help-topic
...2103H-improved-dc/ds-dis
...2103G-Fixed-z-text-trans
...2103G-New-@symoff/on-mac
...2103F-fixed-message-floo
...2103E-fixed-abend-parsin
...2103D-New-jes3-support
...2103D-Changed-list-pgms-
...2103C-new-scroll-positio
...2102a-improved-retrieve-
...2101C-improved-scroll-po
...2101B-diag-server-dumps-
..2020
..2019
...Obsoletescripts-dbc-1907
...Sequencefields-dbc-1907a
...Newseqfoperand-dbc-1907a
...Listread-dbc-1907a
...Readparse-dbc-1907a
...Dbc-1902a
..2018
...Dbc-1811c
..2017
..2016
..2015
...Z21-1505f
..2014
...Z21-1412C
...Z21-1412B
...Z21-1411c
...Z21-1410e
...Z1D-1405B
...Z1D-1405A
..2013
...Z1D-1305b
...Z1D-1302C
...Z1D-1302B
...Z1D-1302A
..2012
...Z1D-1210F
...Z1D-1210E
...Z1D-1210A
...Z1D-1208D
...Z1D-1208A
...Z1D-1207a
...Z1D-1205b
...Z1D-1203b
...Z1D-1202a
...Z1D-1201C
...Z1D-1201B
...Z1D-1201A
..2011
...Z1D-1112c
...Z1D-1111J
...Z1D-1111I
...Z1D-1111H
...Z1D-1111F
...Z1D-1111C
...Z1D-1110M
...Z1D-1110K
...Z1D-1110F
...Z1C-1105C
...Z1C-1105F
...Z1C-1106b
..Support
...Philosophy
...Downloading
...Installing
...Activating
...Removal
...Updatenames
...Blog
.COmmands
..Syntax
...ADdressexpressions
...ASids
...Breakpoints
....Families
....Conditions
.....COUntdown
.....Truefalse
.....COMPArisons
.....COMPOund
.....Null
....Autocmds
...Characterstrings
...Escapecharacter
...NAmes
....Classic
....Uss
...NUmericdata
...Dsnames
...Patternmatch
...Stringdata
....Hexadecimal
....Character
....Mixed
....Decimal
..?
..ADeferred
..ALarm
..ASterisk
..AT
..ATX
..CD
..COMmentary
..CONsole
..COPy
..CUrsor
..DEFine
...DDntrans
....Moreabout
...DSectlib
...Mmefdsn
..DELete
...Cache
...DAtaset
...DDntrans
...DSectlibs
...Equates
...Hooks
...Ilits
...MAPLibs
...MAPS
...MMefdsns
...MOdules
...Proxytasks
..DISConnect
..DISPlay
..DMap
...Load

```
....Csectsandbindermaps
...Clone
..DOwn
..DRop
..DUb
..ENd
..EQuate
...Autocloning
....Example
...Ritargets
..EWhere
..EXec
..FInd
..FOrmat
..FReemain
..GEtmain
..GO
...Nowhere
..GOT
...Nowhere
..GOX
...Nowhere
..HDeferred
..HElp
...Absolute
...Relative
....Examples
...Mixing
..HKeys
..HOok
..IPcs
..ISpf
..Keys
..LEft
..LIBrarylists
...SYntax
...Add
...Checkpoint
....Restore
...DEFault
...DELete
...REDirect
...RESet
...SHow
...Test
....Match
....Try
..LICense
...Panel
...Data
..LISt
...ACCESSLists
...ACCESSRegisters
....Individual
....Registerset
...AL
...ALEs
...AMode
...AR#
...AREgs
...ASID
...AUtomap
...BAng
...BEA
...BELl
...BFr#
...BPts
...BRAnches
...BREakpoints
...CAChe
...CANdet
...CAPs
...CDf
...COLORs
...COLOUrs
...CONSoles
....Mcs
...CONTrolregisters
....Individual
....Registerset
...CR#
...CREgs
...CRH#
...CRHRegs
...CRW#
...CRWRegs
...CUrrent
...CWd
...CXDc
...CXU
...DBc640q
...DDntrans
...DFr#
...DSEctlibs
...DSPaces
...DSTg
...DUB
...DUMp
...DXdc
...EAR#
...EAREgs
...EBea
...ENq
...EPSW
...EPSWE
...EQuates
...ER#
...EREgs
...ERH#
...ERHRegs
...ERW#
...ERWRegs
...ESTaes
....Report
...EXits
...FEatures
....Zos
....Os390
...FIxed
...FLC
...FLOAT
...FLOATIngpointregisters
....Individual
....Registerset
...FOrmat
...FPc
...FR#
...FREgs
...GEneralregisters
....Individual
....Registerset
...GSR#
...GSREgisters
....Individual
....Registerset
...HElp
....Examples
...HFr#
...HILight
...HKeys
...HOoks
...ILC
...ILIt
...INtensity
...ISPf
...ISR@prim
...Keys
...LIcense
...LKedmap
....Classes
....Sections
...LOCAtion
...LOCKs
...LOG
...LOGOnid
...LSEs
...LSTack
...MAIntenance
...MAPLibs
...MAPS
...MEmoryobjects
...MMefdsns
```

```
...MObjects
...MSgs
...NOtes
...NT
...OPErands
...OPTimization
...PARseorder
...PC#
...PEt
...PFkeys
...PGms
....Syntax
....Automaticequates
....Examples
....Report
...PId
...PRIMarysize
...PRINt
...PROfiles
...PSW
...PSWE
...Qualifier
...R#
...RBs
...READ
...READEcho
...REFrprot
...REGIsters
...REGS
...REXx
...RH#
...RHRegs
...RSa
....Types
....Headings
....FLags
....F1sa
....ERrors
....Doc
....EXamples
...RW#
...RWRegs
...SCBs
...SCReen
...SEArchorder
...SECOndarysize
...SECUrity
...SESsions
...SIGnonwait
...SIZeof
...SRbs
...SSCt
...SSRbs
...STATIstics
...STATS
...STEp
...STGlayout
...SUbpools
....Reports
.....Caption
.....Spid
.....Where
.....Used
.....Notes
.....Tcb#
....Examples
...TAsks
...TCbs
...TErminal
...TFs
...TIMeout
...TIOt
...TPut
...TRace
...USErid
...USS
...VARIables
...VARS
...VDisplay
...VEctorregisters
....Individual
....Registerset
...VR#
...VREgs
...VSEttings
...VSTack
...WIndow
...XDc
...XMs
...Zap
..LL
..LOAd
..LOCate
..LOG
..Map
...Syntax
...Bindermaps
...Privatelyloaded
...Assemblermaps
...CMAps
...Dwarffiles
...CMDdialogs
...Findingmapdata
...Examples
..Note
..Off
..PAnelid
..POst
..PRInt
...Help
..PROfile
...REAd
...Save
...RESet
...Omitted
..Quick$init
..REAd
..RECall
..REFresh
..RELease
..RETRIeve
...Usage
...Helpmode
...List
....Editing
....Commands
....Pfkeys
...Examples
..RETRY
..REXx
..RIght
..SCAnlog
..SCRname
..SEt
...ACCessto
....Dataspace
...ASID
...ASN
...AUTOmap
...AUTH
...BAng
...BEll
...BKpt
...BPt
...BReakpoints
...CAndet
...COLORs
...COLOUrs
...CONsole
...CUrrent
...CXdc
...DUB
...DBc640q
...DUMp
...EXits
...FLc
...FOrmat
...HICOLOR
...HICOLOUR
...HILight
...HKeys
...ILC
```

```
...ILIt
...INtensity
...ISPf
...ISR@prim
...Keys
....Onekey
....Keyspanel
....Pfktofkey
....Fkeytopfk
....Special
...LInes
...LOCks
...LOG
....Dataset
....SYsout
....SCrollarea
....Management
...LOGOnid
...Maplibs
...NOBell
...NOHICOLOR
...NOHICOLOUR
...NOIlc
...NOReadecho
...NOWTOR
...Optimization
...PARseorder
...PFkeys
...PRIMarysize
...PRINt
...PROFile
...PROXytasks
...PSW
....AMode
....ASc-mode
....Cc
....Ext
....Io
....Key
....Prog
....State
...PSWE
...Qualifier
...READ
...READEcho
...REFrprot
...REXx
...SCreen
...SECOndarysize
...SECUrity
...SIgnonwait
...STep
...TFs
...TImeout
...TPut
...TRace
....Ignore
....Local
....Roll
....Trap2
...USErid
...USS
...VARIables
...VARS
...VDisplay
...VSettings
...WIndow
....Create
....Delete
....Layouts
....Retrieve
....Vertical
....Horizontal
...WTOR
...Zap
..SHow
...Sizes
...Cdp
...Dbcnnn
..SPLIT
..SPLITV
..STep
..SWAp
..SWItch
..TDeferred
..THaw
..TRACe
...I
...B
...BY
...BNc
...STar
...SB/sa
...SA
...W
..TRAP
..TSo
..UP
..UNdub
..USing
...Dsectref
...Autocloning
...Examples
..Verify
...Targets
...Address
...String
..WAit
..WHEre
..WHItespace
..Zap
...Targets
...Address
...String
...Boolean
..=jump
..%
.CMddialogs
.SHortcutcmds
..ASSocaddr
..?
..A
..Cu
..D
..E
..F
..H
..J
..K
..L
..LL
..LO
..M
..N
..O
..P
..Q
..S
..SH
..T
..W
..X
..Z
..ASTerisk
..|
..\
../
.Pascmds
..Syntax
..Datafields
..Instructions
...F-example
...T-example
.SCripts
..Ourscripts
...AUTOTBnc
...Dcbmaps
...Epmaps
...Ispsetup
...Rsamaps
...SDwamaps
...SPiemaps
```

...STaxmaps
...SVctable
...SYsinfo
...TCbmaps
...TIotmap
...TRapstae
...TSBmaps
...TSOmaps
...XSBmaps
...XSVcmaps
..Ryoscripts
...Sequencefields
...Conditionallogic
...Limitation
.DEbugging
..Gettingstarted
..Amode64
..Batchjobs
..C
...SEtup
...STarting
...Mapping
...Librarylists
....Dwarf
....Csource
....Redirection
....Lists
.....REDirectlist
......Wildcards
......Matchvariables
.....Candidateslists
......Resultobjects
.....RESultslist
.....Families
.....Saving
....Searchorder
....Management
....Glossary
...Debugging
...Variables
...CXDCHook
...CXDCIs
...Externalissues
..CIcs
...Xdccicsx
..Dumps
..EStaes
...Removing
...Superseding
...Modifying
....Informationalcalls
....Example
...Espies
...Hooks
...Debugging
..EXitroutines
...Examples
..Frr
..Hooks
..Ispf
...Panel
....Installing
...DISplays
...DIAlogs
...Authorized
...Xdclbdef
..Jes2
..LE
..LOstlocks
..PCroutines
..PLpa
..REEntrant
...Storagekeys
..REFreshable
...Storagekeys
..RMode64
..Srbmode
..TSo
..Uss
...Operands
....Uss=
....Dub=
...Implementation
...Names
...Xdc
..TIps
...Abendreascd
...Duplicates
....Example
...S0c4
....Estaes
....Badpointers
....Initialregs
....Systemcb
.ADdressing
..Parsers
...Parseorder
...Tags
...Asm
....MIxedcase
....Quoted
....RESolution
.....Searches
.....Examples
....DUplicates
....EXtents
....Longnames
....Asmequs
....Jpalpaplpa
....MOdules
....DSects
....EQuates
....REGisters
....Psws
...Cee
..Numeric
...Scalingcodes
..Dotplus
..Implicit
...Cdpndp
..Targetspaces
..Functions
..Syntax
...BAseterm
....Functions
....Registersandpsws
....Examples
...Offsetterms
....Constants
....Operators
....Functions
....Examples
...BEtweenterms
....Indirect
....Functions
....Examples
..Examples
.FUNctions
..CAtegories
..CHart
..RECursion
..SYntax
..FLc
..ALet
..ASId
..ASN
..F4
..H2
..ILC
..IEq
..ILIt
..INdirect
..Lmod
..NXInst
..NXSeq
..Pclocate
..RDdd
..REAl
..SAve4
..S1-thru-s8
..Width
..XADDr

..XADR
..XALet
..XASId
..XASN
..X1-thru-x8
..ZLeft
..ZRight
.ATtentions
..INXdc
..INUserpgm
..INThebatch
.Breakpoints
..Autocmds
..Conditional
...Watch
..DEAdtraps
...#die
..DEFerred
..Families
..Persistent
..Reentrant
..Stepping-c
...C
....Defaultstep
....Pfkeys
....MIxwithtrace
....MAChinecode
....Autostep
....MAPsrequired
..TRANsient
..TRAPping
..TRACing
...Branches
...STorealter
...SUbroutines
...Alien
...Ex/exrl
...Pfkeys
...Conditional
....Multiple
....Corruptions
....Performance
....Slipvsxdc
....Failures
..TYpes
...Corruption
.CHaractersets
.Hooks
..DYnamic
...Security
...Homespace
...Otherspaces
....Startingnewsessions
...Commonstorage
...Whathappens
..Static
...#xdchook
...Cxdchook
..DEferred
..Legacysupport
..Undoing
.DDnames
..ADAta
..ADUmp
..CDF
..CDFTrace
..CMds
..COff
..Fptnn
..Itrac
..JSTCb
..JSTEp
..Lib
..Maplb
..NComp
..NOAlc
..NOMap
..NOSc
..NStep
..PLib
..PRInt
..PROf
..Quick
..REFr8
..RENt8
..REXx
..SDump
..XClud
..SYSAdata
..SYSCdbg
..SYSMdump
..SYS000nn
..TAsklib
..TLib
..TRAce
..TRSaf
..TSprt
..Wtor
..XDcis
.DUmps
..Startingdumpxdc
...Dxdcproc
....Arguments
...Processingoptions
....Asdefault
....Mmef
....Summarydata
...Switching
...Tsoready
...Batch
..DUmptypes
...Slip
..CONtents
...Foreignvlocal
..DIsagreementswithipcs
...Pswregs
..EXPecteduses
...Personaluse
....Sysmdumps
....C
....Asm
...Isvusage
..MAppingdata
...Bindermaps
...Adatamaps
...Symdatamaps
...DWarfmaps
...Csource
...Literals
...Mmeffiles
...DEfine
...Journaling
...Zoscblocks
..Literals
...Dump/xdc
...Ipcs
..EXEcutionthread
...Initial
...Changing
...Errorandretrylevels
...Srbmode
..COMmands
..MIscellaneous
...DDnames
...Locks
...Profiles
...Closed
...Redacted
...DAtaspaces
.EQuates
..Builtin
...Cblocks
...Storage
...REGSets
...REGIsters
...Internal
...Other
...Automatic
..Floating
..Autocloning
..Ritargets
.EXEcutionlevels

```
..RBs
...Resumeaddress
..Errorlvl
..REtrylvl
...Types
...ESTAEs
...ESTAIs
.EXIts
..Acif
..Frrend
..Init
..Miscxits
..RESume
..Termin8
..UCi
..USErcmds
...Srcxucmd
..USVctrce
..REGstack
...Srcxureg
.FULlscreen
..TErminals
...Geometries
...Ispfsettings
...Colors
...Bindimages
...Problems
..TItle
...Center
...Left
...Right
..Fsinputs
..Displayerrs
..Ispf
..Logging
...Latentcommands
..Notes
..PFkeys
...HYphen
...Underscore
...Pfkeysets
....Suggestions
...Defaultkeys
....SET-A
....SET-B
...HElpkeys
..Retrieve
..SCrolling
...Storagescrolling
..Windows
...Primarywindow
...WAtchwindows
.MACros
..#Cblocks
..#DBCParm
..#DBCR0In
..#DBCR0Ut
..#DBCVrsn
..#DIe
..@symoff/on
..#Test
..#XDCHook
..#XDCLoc8
.MAPs
..AUTOMapping
..Bindermaps
..CSECTMaps
...Asmoffsets
..DSectmaps
..CMaps
..CSECTSasdsects
..Privatelyloaded
...Dmapandusing
...Map
...Ussfiles
..Sym
...Excesssymdata
....Examples
....Xdcsymed
....Tachyon
..AData
...CReating
...MAPlibs
....Namedlists
...Searchorder
...CAching
...Performance
...Excessadata
....Examples
....Xdcadata
...MACros
...Tips
..DWarf
..Floating
..AUTOCloning
..Xdcmaps
...Adata
...Yourdoc
..CIcsmaps
.MEssages
..DBC001
..DBC002
..DBC003
..DBC004
..DBC005
..DBC006
..DBC007
..DBC008
..DBC009
..DBC010
..DBC011
..DBC012
..DBC013
..DBC014
..DBC015
..DBC016
..DBC017
..DBC018
..DBC019
..DBC020
..DBC021
..DBC022
..DBC023
..DBC024
..DBC025
..DBC026
..DBC027
..DBC028
..DBC029
..DBC030
..DBC031
..DBC032
..DBC033
..DBC034
..DBC035
..DBC036
..DBC037
..DBC038
..DBC039
..DBC040
..DBC041
..DBC042
..DBC043
..DBC044
..DBC045
..DBC046
..DBC047
..DBC048
..DBC049
..DBC050
..DBC051
..DBC053
..DBC054
..DBC055
..DBC056
..DBC057
..DBC058
..DBC059
..DBC060
..DBC061
..DBC062
..DBC063
```

```
..DBC064
..DBC065
..DBC066
..DBC067
..DBC069
..DBC070
..DBC071
..DBC072
..DBC073
..DBC074
..DBC075
..DBC076
..DBC077
..DBC078
..DBC079
..DBC080
..DBC081
..DBC082
..DBC083
..DBC084
..DBC085
..DBC086
..DBC087
..DBC088
..DBC089
..DBC090
..DBC091
..DBC092
..DBC093
..DBC094
..DBC095
..DBC096
..DBC097
..DBC098
..DBC099
..DBC100
..DBC101
..DBC102
..DBC103
..DBC104
..DBC105
..DBC106
..DBC107
..DBC108
..DBC109
..DBC110
..DBC111
..DBC112
..DBC113
...Reasons
..DBC114
..DBC115
..DBC116
..DBC117
..DBC118
..DBC119
..DBC120
..DBC121
..DBC122
..DBC123
..DBC124
..DBC125
..DBC126
..DBC127
..DBC128
..DBC129
..DBC130
..DBC131
..DBC132
..DBC133
..DBC134
..DBC135
..DBC136
..DBC137
..DBC138
..DBC139
..DBC140
..DBC141
..DBC142
..DBC143
..DBC144
..DBC145
..DBC146
..DBC147
..DBC148
..DBC149
..DBC150
..DBC151
..DBC152
..DBC153
..DBC156
..DBC157
..DBC158
..DBC159
...19-24
...34-42
..DBC160
..DBC161
..DBC162
..DBC163
..DBC164
..DBC165
..DBC166
..DBC167
..DBC168
..DBC169
..DBC170
..DBC171
..DBC172
..DBC173
..DBC174
..DBC175
..DBC176
..DBC177
..DBC178
..DBC179
..DBC180
..DBC181
..DBC182
..DBC183
..DBC184
..DBC185
..DBC186
..DBC187
..DBC188
..DBC189
..DBC190
..DBC191
..DBC192
..DBC193
..DBC194
..DBC200
..DBC201
..DBC202
..DBC203
..DBC204
..DBC205
..DBC206
..DBC207
..DBC208
..DBC209
..DBC210
..DBC211
..DBC212
..DBC213
..DBC214
..DBC215
..DBC216
..DBC217
..DBC218
..DBC300
..DBC301
..DBC302
..DBC303
..DBC304
..DBC305
..DBC306
..DBC307
..DBC308
..DBC309
..DBC310
..DBC311
```

..DBC312
..DBC313
..DBC314
..DBC315
..DBC316
..DBC317
..DBC318
..DBC319
..DBC320
..DBC323
..DBC324
..DBC340
..DBC341
..DBC343
..DBC344
..DBC345
..DBC346
..DBC347
..DBC348
..DBC349
..DBC350
..DBC351
..DBC352
..DBC353
..DBC354
..DBC355
..DBC356
..DBC357
..DBC358
..DBC359
..DBC360
..DBC361
..DBC362
..DBC363
..DBC364
..DBC365
..DBC366
..DBC367
..DBC368
..DBC369
..DBC370
..DBC371
..DBC372
..DBC373
..DBC374
..DBC375
..DBC376
..DBC377
..DBC378
..DBC379
..DBC380
..DBC381
..DBC382
..DBC383
..DBC384
..DBC385
..DBC386
..DBC387
..DBC388
..DBC389
..DBC391
..DBC392
..DBC393
..DBC394
..DBC395
..DBC396
..DBC397
..DBC398
..DBC399
..DBC400
..DBC401
..DBC402
..DBC403
..DBC404
..DBC405
..DBC406
..DBC407
..DBC408
..DBC409
..DBC410
..DBC411
..DBC412
..DBC413
..DBC414
..DBC415
..DBC416
..DBC450
..DBC451
..DBC452
..DBC494
..DBC495
..DBC496
..DBC497
..DBC498
..DBC499
..DBC500
..DBC501
..DBC502
..DBC503
..DBC504
..DBC505
..DBC506
..DBC507
..DBC508
..DBC509
..DBC510
..DBC511
...Reasons
..DBC512
..DBC513
..DBC514
...ACif
...ANchor
...Clonename
...DBCcvt
...DBPtsupport
...HOOKMisc
...HOOKSvc
...Pcroutines
...QUICKXDc
...QUICKXXx
...SErvicesvc
...SSCt
...SSExits
...SVc61
...SRbbundle
...TImerdie
...TRap
...Updatelevel
...STatus
...Failures
..DBC515
..DBC516
..DBC517
..DBC518
..DBC519
..DBC520
..DBC521
..DBC522
..DBC523
..DBC524
..DBC525
..DBC526
..DBC527
..DBC528
..DBC529
..DBC530
..DBC531
..DBC532
..DBC533
..DBC534
..DBC535
..DBC536
..DBC537
..DBC538
..DBC539
..DBC540
..DBC541
..DBC542
..DBC543
..DBC544
..DBC545

..DBC546
..DBC547
..DBC548
..DBC549
..DBC550
..DBC551
..DBC552
..DBC553
..DBC554
..DBC555
..DBC556
..DBC557
..DBC558
..DBC559
..DBC560
..DBC561
..DBC562
..DBC563
..DBC564
..DBC565
..DBC566
..DBC567
..DBC568
..DBC569
..DBC570
..DBC571
..DBC572
..DBC573
..DBC574
..DBC575
..DBC576
..DBC577
..DBC578
..DBC579
..DBC580
..DBC581
..DBC582
..DBC583
..DBC584
..DBC585
..DBC586
..DBC587
..DBC588
..DBC589
..DBC590
..DBC591
..DBC592
..DBC593
..DBC594
..DBC595
..DBC596
..DBC597
..DBC598
..DBC599
..DBC600
..DBC601
..DBC602
..DBC603
..DBC604
..DBC605
..DBC606
..DBC607
..DBC608
..DBC609
..DBC610
..DBC611
..DBC612
..DBC613
..DBC614
..DBC615
..DBC616
..DBC617
..DBC618
..DBC619
..DBC620
..DBC621
..DBC622
..DBC623
..DBC624
..DBC625
..DBC626
..DBC627
..DBC628
..DBC629
..DBC630
..DBC631
..DBC632
..DBC633
..DBC634
..DBC635
..DBC636
..DBC637
..DBC638
..DBC639
..DBC640
..DBC641
..DBC642
..DBC643
..DBC644
..DBC645
..DBC646
..DBC647
..DBC648
..DBC649
..DBC650
..DBC651
..DBC652
..DBC653
..DBC654
..DBC655
..DBC656
..DBC657
..DBC658
..DBC659
..DBC660
..DBC661
..DBC662
..DBC663
..DBC664
..DBC665
..DBC666
..DBC667
..DBC668
..DBC669
..DBC670
..DBC671
..DBC672
..DBC673
..DBC674
..DBC675
..DBC676
..DBC677
..DBC678
..DBC679
..DBC680
..DBC681
..DBC682
..DBC683
..DBC684
..DBC685
..DBC686
..DBC687
..DBC688
..DBC689
..DBC690
..DBC691
..DBC692
..DBC693
..DBC694
..DBC695
..DBC696
..DBC697
..DBC698
..DBC699
..DBC700
..DBC701
..DBC702
..DBC703
..DBC704
..DBC705
..DBC706
..DBC707

..DBC708
..DBC709
..DBC710
..DBC711
..DBC712
..DBC713
..DBC714
..DBC715
..DBC716
..DBC717
..DBC718
..DBC719
..DBC720
..DBC721
..DBC722
..DBC723
..DBC724
..DBC725
..DBC726
..DBC727
..DBC728
..DBC729
..DBC730
..DBC731
..DBC732
..DBC733
..DBC734
..DBC735
..DBC736
..DBC737
..DBC738
..DBC739
..DBC740
..DBC741
..DBC742
..DBC743
..DBC744
..DBC745
..DBC746
..DBC747
..DBC748
..DBC749
..DBC750
..DBC751
..DBC752
..DBC753
..DBC754
..DBC755
..DBC756
..DBC757
..DBC758
..DBC759
..DBC760
..DBC761
..DBC762
..DBC763
..DBC764
..DBC765
..DBC766
..DBC767
..DBC768
..DBC769
..DBC770
...Reasons
..DBC771
..DBC772
..DBC773
..DBC774
..DBC775
..DBC776
..DBC777
..DBC778
..DBC779
..DBC780
..DBC781
..DBC782
..DBC783
..DBC784
..DBC785
..DBC786
..DBC787
..DBC788
..DBC789
..DBC790
..DBC791
..DBC792
..DBC793
..DBC794
..DBC795
..DBC796
..DBC797
..DBC798
..DBC799
..DBC800
..DBC801
..DBC803
..DBC804
..DBC805
..DBC806
..DBC807
..DBC808
..DBC809
..DBC810
..DBC811
..DBC812
..DBC813
..DBC814
..DBC815
..DBC816
..DBC817
..DBC818
..DBC820
..DBC821
..DBC822
..DBC823
..DBC824
..DBC825
..DBC826
..DBC827
..DBC828
..DBC829
..DBC830
..DBC831
..DBC832
..DBC833
..DBC834
..DBC835
..DBC836
..DBC837
..DBC838
..DBC839
..DBC840
..DBC841
..DBC842
..DBC843
..DBC844
..DBC845
..DBC846
..DBC847
..DBC848
..DBC849
..DBC850
..DBC851
..DBC852
..DBC853
..DBC854
..DBC855
..DBC856
..DBC857
..DBC858
..DBC859
..DBC860
..DBC861
..DBC862
..DBC863
..DBC864
..DBC865
..DBC866
..DBC867
..DBC868
..DBC869
..DBC870

..DBC871
..DBC873
..DBC874
..DBC875
..DBC876
..DBC877
..DBC878
..DBC879
..DBC880
..DBC881
..DBC882
..DBC883
..DBC884
..DBC885
..DBC886
..DBC887
..DBC888
..DBC889
..DBC890
..DBC891
..DBC892
..DBC893
..DBC894
..DBC895
..DBC896
..DBC897
..DBC898
..DBC899
..DBC900
..DBC901
..DBC902
..DBC903
..DBC904
..DBC905
..DBC906
..DBC907
..DBC908
..DBC909
..DBC910
..DBC911
..DBC912
..DBC913
..DBC914
..DBC915
..DBC916
..DBC917
..DBC918
..DBC919
..DBC920
..DBC921
..DBC922
..DBC923
..DBC924
..DBC925
..DBC926
..DBC927
..DBC928
..DBC929
..DBC930
..DBC931
..DBC932
..DBC933
..DBC934
..DBC935
..DBC936
..DBC937
..DBC938
..DBC939
..DBC940
..DBC941
..DBC942
..DBC943
..DBC944
..DBC945
..DBC946
..DBC947
..DBC948
..DBC949
..DBC950
..DBC951
..DBC952
..DBC953
..DBC954
..DBC955
..DBC956
...S106-0e
...S306-0c
...S306-20
...S306-30
...S806-04
..DBC957
..DBC958
..DBC959
..DBC960
..DBC961
..DBC962
..DBC963
..DBC964
..DBC965
..DBC966
..DBC967
..DBC968
..DBC969
..DBC970
..DBC971
..DBC972
...License
..DBC973
..DBC974
..DBC975
..DBC976
..DBC977
..DBC978
...CUstomer
...STartdate
...Enddate
...Warning
...COntrol
...Model
...Permit
...SErial
..DBC979
..DBC980
..DBC981
..DBC982
..DBC983
..DBC984
..DBC985
..DBC986
..DBC987
..DBC988
..DBC989
..DBC990
..DBC991
..DBC992
..DBC993
..DBC994
..DBC995
..DBC996
..DBC997
..DBC998
..DBC999
..DBC1000
..DBC1010
..DBC1020
..DBC1030
..DBC1100
..DBC1101
..DBC1102
..DBC1103
..DBC1111
..DBC1112
..DBC1113
..DBC1114
..DBC1121
..DBC1122
..DBC1123
..DBC1124
..DBC1129
..DBC1130
..DBC1150
..DBC1151

```
..DBC1152
..DBC1153
..DBC1159
..DBC1160
..DBC1170
..DBC1180
..DBC1181
..DBC1182
..DBC1183
..DBC1185
..DBC1187
..DBC1188
..DBC1189
..DBC1210
..DBC1600
..DBC1601
..DBC1602
..DBC1603
..DBC1604
..DBC1605
..DBC1606
..DBC1620
..DBC1621
..DBC1630
..DBC1631
..DBC1632
..DBC1633
..DBC1634
..DBC1635
..DBC1636
..DBC1637
..DBC1640
..DBC1641
..DBC1642
..DBC1650
..DBC1651
..DBC1652
..DBC1653
..DBC1661
..DBC1662
..DBC1663
..DBC1664
..DBC1670
..DBC1680
..DBC1681
..DBC1682
..DBC1687
..DBC1688
..DBC1689
..DBC1700
..DBC1800
..DBC1801
..DBC1802
..DBC1803
..DBC1804
..DBC1805
..DBC1806
..DBC1807
..DBC1808
..DBC1809
..DBC1820
..DBC1821
..DBC1830
..DBC1831
..DBC1832
..DBC1833
..DBC1860
..DBC1861
..DBC1862
..DBC1863
..DBC1865
..DBC1866
..DBC1867
..DBC1868
..DBC1870
..DBC1871
..DBC1872
..DBC1900
..DBC1901
.MUltitask
..Scope
..Conversationmanagement
..Irbs
.PRofiles
..MEnu
...TSoispfcdfcics
...TErmsettings
...COlors
...Pfktofkeymap
...Keys
...Hkeys
...Log
...Windows
...REAdzapparse
....SToreprotzap
....Mnemoniczap
....Readecho
....SEqfield
....SCriptlibname
....Parseorder
...Formattracefind
....SOurceformatting
....Macroexpansions
....FOrmattingbias
....SToragelocations
....DISPLACementfields
....TExtencoding
....DISPLAYwidths
....Pointerresolutions
....TRACEPositioning
....TRACEBasr
....Breakpointopcode
....DEadtrapbypass
....FIndformatting
...Automapprintetc
....Automap
....Lpp
....SYsoutclass
....Holdnohold
....CAnceldetachdebugging
....SEtbang
....COnversationtimeout
....Flc
...REXx
...CXdcsettings
...Uss
..MApliblists
..Readsave
..Namedprofiles
...Example
..Profnames
..DEfaultprofile
..Factorydefaults
...A80
...AWide
...C80
...CWide
..Oldprofiles
..DDnames
..Tlibprofiles
..Wide
.Rexx
..Environment
..SAYoutput
..Input
..XDCServices
...XDCCmd
...XDCDump
...XDCGet
...XDCMemry
...XDCParse
...XDCRDisp
...XDCROute
...XDCSAy
...XDCSTate
...XDCVars
...XDCZAp
...XDCZStat
..Userfunctions
..XDCCommands
..SAMpleexecs
.SEcurity
```

..Rules
..AUth
..CDf
..Fasm
..Lostlocks
..Zap
...Description
...Oldrules
...Suggestions
..Gzap
..ACif
..CAche
..Trace
.Tachyon
.USERInterfaces
..Fullscreen
..Cdf
..Linemode
..Wtowtor
..Uci
.UTilities
..XDCAdata
..XDCCall
..XDCMmef
..XDCSymed
.Virtmem
..Concepts
..ACcesslists
..ALets
..Xdcaccess
...Fasm
....Real
...Lasm
..Mvsdoc
.XDCCAll
..ENvironment
..Interferingestaes
..EXits
..Tasklib
...Tasklibraries
..Jobsteptasks
..Ddnames
.XDCCLones
..Using
..Axdcis
..Cxdcis
..Profiles
.XDCSrver
..Isbouncingsafe?
..Cdf
...Setup
...COMMUnications
...CONnection
...Tso
...Logon
...COMMAnds
...Fasm
...Recovery
...CRosssysplex
.SUpport
..Featuresandcaps
..Hlq
..Manuals
..Online
...Website
...Facebook
...Youtube
..COntactus
..SEndingusdumps
..SUpportstatement
..Legalnotice
..CRedits
.Whatsnew
..Z22
...Autostepping
...Builtinhelp
...CIcs
...COmmands
...CXdc
...DDnames
...DUmps
...Equates
...Frr
...MAPpingtheunmappable
...POintandshoot
...Newprocedure
...PROFILEMenuingsystem
...PROFILEResetsanddefaults
...Rexx
...SHortcutcmds
...STartuppanel
...Trap2
...Incompatibilities
....Exrl
....Autocmdstrings
..Z21
...AUTOMapping
...AUTOSCrolling
...AUTOSTepping
...BEa
...BUiltinhelp
...Commands
...Equates
...Frr
...HElpcrosslinks
...HOokrewrite
...MAplibslist
...ONlinehelp
...OPerandvalues
...PArseorder
...PROFILEMenuingsystem
...PROFILEResetsanddefaults
...SHortcutcmds
...SUpportimprovements
...SYntaxchanges
...TAggedaddressexpressions
...Watchwindowdefault
...MIscellaneous
...THingsfixed
...Incompatibilities
..Z113
...ABovethebarpgms
...ADdressexpressions
...Commands
...Equates
...Hooks
...Builtinhelp
...Zosr13support
...Miscellaneous
...Incompatibilities
....#psw
....Lkedjcl
..Z112
...Addressexpressions
...COmmands
...CDf
...Dataspaces
...Equates
...Functions
...CMddialogs
...Builtinhelp
...Pointandshoot
...SCriptsupport
...SRbsupport
...Zenterprise
...Miscellaneous
....Embeddedlabels
....Offsetsdisplay
....Sysinfo
....Windowscrolling
....Bugsfixed
...Incompatibilities
....AReaequates
....ASpacerefs
....ATtentions
....CDf
....COLors
....COMmentstrings
....Gocommand
....LArgedisplays
....LOnequestionmarks
....REAdecho

....REGptrs
....Systemshutdown
....XDCAcif
....XDCMapschanges
..Z110
...Addresses
...BUILTINEquates
...Commands
...Ddnames
...Features
...BUILTINHelp
...SCripts
...STartuppanel
...Z10processor
...Miscellaneous
....Controlregisterdisplays
....Frrproxytasksmoved
....Hd0c4fixed
....Licensecontroldata
....Problemstatepc
....Rexxenvironment
....#xdcismacro
...Incompatibilities
....Builtinfunctions
....Rexxenvironment
....SUbpoolusage
....SVcrequired
....Userexits
....Xdccall
..Z19
...BFp
...BRanches
...Commands
...DDnames
...DFp
...BUiltinhelp
...Profiles
...REFrprot
...REXx
...Scripts
...Zosr19
...Miscellaneous
...Incompatibilities
..Z18
...AData
...ARtifacts
...BReakpoints
...CDf
...COMmands
...CONsoles
...DDnames
...DUmps
...Equates
...FOrmatting
...FRr
...MACros
...MAPlibs
...BUiltinhelp
...POintandshoot
...PRofiles
...Quickstart
...SEcurity
...SRbmode
...SToragescrolling
...Xdccall
...Z9
...MIscellaneous
...Incompatibilities
....Danelen
....Exits
....Find
....PFkeys
....POintandshoot
....Retrieve
....SECurity
....SETbang
....SHow
....XDCAcif
....XDCPanel
..Z16
...AData
...ALrf
...Commands
...Exits
...Xdcmaps
...Incompatibilities
...Miscellaneous
..Z13
...Autocloning
...CAsesensitivity
...COmmands
....DEletemaps
....DMap
....Equate
....FInd
....FOrmat
....Indirect
....LISTLked
....LISTMap
....Map
....SEtformat
....SHow
....Using
....Where
...Equates
...Help
...MApdisplays
...Programobjects
...Scripts
...MIscellaneous
..Z12
...Zarchitecture
....Storage
....Registers
....Instructions
...Ldf
...CDf
...COmmands
....Syntax
.....Asids
.....Characterstrings
....New
.....Got
.....LISTASID
.....LISTASIS
.....LISTBang
.....LISTMemoryobjects
.....LISTRegs
.....LISTStatistics
.....Panelid
.....SCrname
.....SEtbang
.....Null
....Changed
.....Alarm
.....Copy
.....DEleteequates
.....DIsplayformat
.....Equate
.....Find
.....Go
.....Keys
.....LISTFeatures
.....LISTPsw
.....LISTREgs
.....LISTRN
.....LISTTAsks
.....LISTTRace
.....Off
.....SETBkpt
.....SETLog
.....SETMaplibs
.....SETPsw
.....SETTrace
.....SHOWWhere
.....SHOW
.....Using
....Deleted
...DEferredbreakpoints
...DIsplays
....Adata
...Equates

...FULlscreen
...FUNctions
...Incompatibilities
....Sdwausage
...MUltitask
...Builtinhelp
...Profiles
....Finddisplay
....Setbang
....Tbydeadtraps
...Scripts
...Tracing
...Widedisplays
...Xdccall
...MIscellaneous
..S20M
..S20
...Breakpoints
....Conditional
....Disable
....Storealter
....Watch
....X
...Commands
....SHortcutcmds
....Deleted
....SCripts
...Hookprocessing
...Languageenvironment
...SCrolling
...SOurcelevel
...Miscellaneous
..S10
...ADdressing
....Builtin
....Decimal
...ARr
...Commands
....Copy
....End
....Find
....LCap
....LLkedmap
....LPgms
....LSessions
....LTasks
....SASId
....SASN
....SLOG
....SLOGDsn
....SPswcc
....SHow
....Verify
....Dsnames
...DBcparm
...DSnames
...Help
...LOGdisk
...LONgnames
...Opcodes
...Pdse
...Tracing
....Cdp
....Numfunctions
....Pfkeys
...Usagepricing
...Xms
...Miscellaneous
..X33
...Commands
....Alarm
....Commentary
....LCDF
....LCDFBatch
....LMaintenance
....LVtam
....LXdc
....SAlarm
....SBell
...Equs
...INstructions
...IRbs
...ISpf
...Longnames
...MIxedcase
...MUltitask
...Builtinhelp
...Subscripts
...Tachyonsoft
...Vscr
...Xdcsymed
...Year2000
..X32
...Ascii
...BUILTINEqu
...CDf
...CMdfiles
...COmmands
....DHooks
....DIsplay
....End
....FInd
....FOrmat
....Getmain
....Hook
....LCandet
....LEstae
....LHooks
....LIntensity
....LLSTack
....LLSEs
....LPrint
....LScbs
....LTCbs
....LTIot
....Phelp
....SCandet
....SIntensity
....SKeysreset
....SPRInt
....SPROfile
....SHow
....Verify
....Where
....Zap
...DEcimaldata
...Exits
...FAsm
...FOrmatting
...Hooks
...SHortcutcmds
...MEssages
...BUILTINHelp
...PRIntingmanuals
...PROfile
...SEcurity
...MIscellaneous
...DIfferences
..X30
...Addrexp
...BReakpoints
...BUILTINEqu
...CDf
...CHngedcmds
...CIcs
...CMdfiles
...Exits
...FLoatequ
...FOrmatter
...License
...SHortcutcmds
...MEmcmds
...MIsc
...MVsesa
...NEwcmds
...NOtes
...BUILTINHelp
...PFkeys
...PRofiles
...REAlmem
...RETrieve
...RPlcedcmds

...SOurcelvl
...SYmedit
...Windows
...Xdcif

Help

About Z/XDC Help

Z/XDC HELP information is vast and comprehensive. There are nearly three thousand topics of information organized into a hierarchy that hopefully makes sense.

The Help topics are available in three forms:

Built-in - All topics can be accessed within Z/XDC using three commands and a shortcut: **HELP, LIST HELP, PRINT HELP** and **H.**

Online - All topics can be accessed externally on the web at **colesoft.com/help.**

PDFs - All topics are also available in PDF form at **colesoft.com/pdfs.**

For more information, select one of the crosslinks above. (If you're using the Built-in form of the Help, then enter **H** into the shortcut fields at the left above.)

Important! When using the Built-in form of the Help, there are significant differences between the way that Help works, and the way other help systems may work. These differences may take some getting used to, so please read **HELP ABOUTHELP BUILT-IN** for the details.

Product Library Names

Throughout the Help, Product Library names are referred to as **hlq.XDCV31.XDCwhatever,** where:

- **hlq** is the library name's High Level Qualifier. Every customer will create the Z/XDC Product Libraries using their own, in-house High Level Qualifier. You will have to ask your SysProg what the hlq is set to at your site.

- **whatever** is a library's unique name.

For more information, see **HELP SUPPORT HLQ.**

Topics Index

For information on the following topics, type an **H** in the shortcut fields below at the left. Then press ENTER.

To show the crosslinks without jumping to the linked-to text, type a **?** (question mark) in the shortcut field.

You may use **UP** and **DOWN** commands **(F7** and **F8)** to scroll up or down this or any Built-in Help topic.

You may use **HELP *NEXT** and **HELP *PREVIOUS F11** and **F10)** to proceed sequentially forwards or backwards from one topic to the next.

Refer to the PF key definitions displayed at the bottom of the screen for more information.

The **hilighted** topics below are, perhaps, the ones you might want to review first.

TOPIC	DESCRIPTION
ABOUTHELP	- About the structure of Z/XDC's Help, about the three forms of the Help, and about how to navigate the Help in each of its three forms.
MAINTENANCE	- Maintenance and New Functionality Guide **All new things are described here first!**
COMMANDS	- Z/XDC command descriptions.
CMDDIALOGS	- Comprehensive dialogs to help issue various of Z/XDC's more complex commands.
SHORTCUTCMDS	- 1- and 2-character commands accepted by Shortcut Entry Fields.
PASCMDS	- Short Point-and-Shoot commands that you can set onto machine instructions or data fields to display or set traps at the locations those fields or instructions point to.
SCRIPTS	- A library of useful Z/XDC command scripts.
DEBUGGING	- Debugging various types of programs: - **Assembler** programs from the simplest to the most complex - **XL C/C++ and Metal C programs** - **Multitasking, PC** and **SVC** routines - **Reentrant** and **refreshable** routines - **SRBs** and **FRR protected** task mode routines - **Batch jobs, started tasks** and **subsystems** - System and vendor exit routines - Routines located in Common Storage, even in the **PLPA** - **AMODE64** and even **RMODE64** code - TSO and ISPF commands, programs and dialogs - Programs executing in a **USS** (OMVS) environment - **Authorized** and non-authorized programs alike
ADDRESSING	- Virtual storage references.
FUNCTIONS	- Built-in functions for use in address expressions, conditional breakpoints, the VERIFY command, and elsewhere.
ATTENTIONS	- Attention processing in Z/XDC.
BREAKPOINTS	- Setting traps and WATCHs and performing execution traces.
CHARACTERSETS	- Character sets used by Z/XDC.
HOOKS	- Creating **on the fly** debugging sessions in tasks, SRBs and other routines running anywhere.
DDNAMES	- A description of all ddnames (keyword and otherwise) used by Z/XDC.
DUMPS	- using Z/XDC to examine IPCS dumps.
EQUATES	- Automatically defined equate names.
EXECUTIONLEVELS	- Error level and retry level execution environments. **Understanding this topic is important!**
EXITS	- Exit routines supported by Z/XDC.
FULLSCREEN	- Fullscreen Support: Windows, zapping, PF keys, session scrolling

and logging, session profiles, etc.

MACROS	- Z/XDC Assembler macros that customers may find useful to use in their own programming.
MAPS	- Characteristics and considerations of symbol tables and source image maps.
MESSAGES	- Z/XDC information and error messages.
MULTITASK	- Multi-tasking debugging.
PROFILES	- Creating and managing user profiles.
REXX	- REXX commands support.
SECURITY	- An explanation of the "access control" rules that Z/XDC checks to control potential system integrity violations.
TACHYON	- Advantages of using Tachyon Software's **z/Assembler** in conjunction with Z/XDC.
USERINTERFACES	- 3270, browser [planned], line mode, and other communication interfaces used by Z/XDC.
UTILITIES	- Additional utility programs related to Z/XDC.
VIRTMEM	- A discussion or address spaces and data spaces, and Z/XDC's capabilities regarding them.
XDCCALL	- Using XDCCALL, XDCCMD, XDCCALLA, and XDCCMDA.
XDCCLONES	- Using a renamed XDC.
XDCSRVER	- All about **server/XDC,** what it is and what it does.
SUPPORT	- ColeSoft wants to make sure that you get good use from Z/XDC. If you are unable to answer your questions by consulting the available documentation and this Built-in Help, please send an email to **support@colesoft.com.** For further details, see **HELP SUPPORT** both for complete contact information, and for instructions on **how to send us dumps.**

Help ABouthelp

About Z/XDC Help

Z/XDC HELP information is vast and comprehensive. There are nearly three thousand topics of information organized into a hierarchy that hopefully makes sense.

Overall Structure

The Built-in Help topics are organized into a hierarchy with a Master Index topic (named **HELP)** at the top, major topics one level down, and various subtopics below that. Illustration:

```
          A          <-- Master Index
         /|\
        / | \
       B  E  F       <-- Topics
      /|    / \
```

```
     / |   /   \
    C  D  G     K    <-- Subtopics
         /|\
        / | \
       H  I  J       <-- Etc.
```

Natural Topic Pathways

A number of "natural" pathways occur through the structure...

The sisters path: This is a horizontal path runs from one **sister** topic to the next
In the above illustration, the following are sister paths:

    A
    B-E-F
    C-D
    G-K
    H-I-J

Note, sister paths do not jump to cousins.

Generally, whenever a Built-in Help topic shows a list of subtopics, each subtopic is a "daughter" topic of the current topic, so each subtopic also is a "sister" topic to all of the other subtopics in that same list.

The parent-child path: This is a vertical path that runs up and down between ancestor/descendant topics. In the above illustration, the following are parent-child paths:

    C-B-A
    D-B-A
    E-A
    H-G-F-A
    I-G-F-A
    J-G-F-A
    K-F-A

The depth-first sequential path: This is a path that reaches **every** topic in the database. It starts at the top topic and descends leftwards where possible and moves horizontally or rises rightwards otherwise. In the above illustration, the following is the depth-first sequential path:

    A-B-C-D-E-F-G-H-I-J-K

Every topic in the database has a name that is unique among its sisters but not necessarily unique in the database. For example, there are several topics named **EXAMPLES** in the database, but in no case will there be two sister topics named **EXAMPLES.**

Where is the Help?

The Help topics are available in three forms:

BUILT-IN - All topics can be accessed within Z/XDC using three commands and a shortcut:
- **HELP**
- **LIST HELP**
- **PRINT HELP**
- **H**

See **HELP ABOUTHELP BUILT-IN** for details.

ONLINE - All topics can be accessed externally on the web at **colesoft.com/help.**

PDFS - All topics are also available in PDF form at **colesoft.com/pdfs.**

For more information, select one of the crosslinks above. (If you're using the Built-in form of the Help, then enter **H** into the shortcut fields at the left.)

Accessing Z/XDC's Help

Within Z/XDC, during an active debugging session, you can use several commands **(HELP, LIST HELP, PRINT HELP)** and shortcuts **(H)** to access either topics that interest you or that relate to what is going on in the session. However, these built-in mechanisms access only the built-in form of the help. See **HELP ABOUTHELP BUILT-IN** for the details.

To access the online form of the help, you need to use your favorite web browser to navigate to **colesoft.com/help.** There is no direct connection within a debugging session. Z/XDC does not have hot links to the web. If it does show you a URL, you have to use copy/paste to use it.

Similarly, if you wish to download the PDF form of the help, you will have to navigate your browser to **colesoft.com/pdfs** to get them. There is nothing within Z/XDC that automates this.

Using Cross Topic Links

A Help topic can be selected by **crosslinks** from other Built-in Help topics. A very large number of crosslinks exist throughout the Help database. A crosslink connects a paragraph in one topic to some other topic that contains information that is related to or is more descriptive of the subject being discussed in that paragraph.

When a crosslink is present, a **Shortcut Entry Field** is displayed to the left of the paragraph containing the crosslink.

In the **built-in form** of the Help, the crosslink is rendered as an underscore [_]:
- To use it, place an **H** or **S** (synonyms) into it, and press **ENTER.**
- To view the crosslink, place a **?** into the shortcut field.

In the **Online form** and the **PDF form** of the Help, the shortcut is rendered as an icon you can click on.

In the **Built-in form,** you can use the **RETRIEVE** command (function key **F12)** to return

to the prior topic.

In the **Online form,** you can use the browser's **back** button.

Viewing the Complete List of Built-in Help Topics

To view the names of **all** of the topics in Z/XDC's Help (Yeah. All 2,775 of them [give or take]):

- In the Built-in form, issue **LIST HELP 9.**
- In the Online form, use the Navigation sidebar at the left.
- In the PDF form, check out the tables of contents.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

CONVENTIONS - A discussion of the "meta-language" of the syntax descriptions shown in the various Built-in Help displays

PRINTING - How to print plain text copies of Z/XDC manuals (not PDFs) from the Built-in Help database

Help ABouthelp Built-in

This topic is about the **Built-in** form of Z/XDC's Help.

Altogether, there are three forms of the Help, all containing the same content:

- **Help Online** is available online at **colesoft.com/help.** It can be accessed outside of a debugging session using just a browser. Also, it is searchable.

- The **Help PDFs** also are available online at **colesoft.com/pdfs.** They can be downloaded and archived on your desktop.

- **Built-in Help** is built into Z/XDC itself and is directly accessible from a debugging session via three commands and a shortcut: **HELP, LIST HELP, PRINT HELP** and **H.**

To learn about the structure and organization of Z/XDC's Help, please read my parent topic, **HELP ABOUTHELP.**

To learn about the commands and methods for accessing the Built-in form of the Help topics, please continue reading this topic.

Using the Built-in Form of the Help

Within Z/XDC, during an active debugging session, you can use the following commands and shortcuts to access the **Built-in** form of the Help:

- Use the **HELP** command to select a topic to read. **HELP** brings up a topic in a scrollable and searchable display.
- Use the **LIST HELP** command to navigate the Help's Topics Index.
- Use the **PRINT HELP** command to print selected Help Topics as text files (not as PDFs).
- Use the **H** shortcut command to select topics that are related to some message or report. Specifically:
  - The **LIST HELP** report messages accept **H** and **S** shortcuts (synonyms) to display the particular topic that a report message names.
  - All **DBCnnn** error messages accept **H** shortcuts to display the message's Built-in Help.
  - **Help topics** themselves contain thousands of crosslinks that connect the current discussion to related topics located elsewhere. These crosslinks accept either **H** or **S** shortcuts (synonyms) to access the related topic.
  - When you finish reading a topic, you can return to your prior topic using the **RETRIEVE** PF key [factory default **F12].**

Navigating Within the Built-in Help

In general, a Built-in Help topic can be selected for display within a debugging session by the following methods:

Absolute References:

A Help topic can be selected by direct reference (also called **absolute reference).** A topic's absolute reference is its name. For more information about **ABSOLUTE** references, please see **HELP COMMANDS HELP ABSOLUTE.**

For **every** topic in the database, there exists a **direct path** to it, downwards from the Help's Master Index topic (the **HELP** topic). This direct path is the names of all topics from the master topic to the desired topic. Examples:

- The direct path to the current topic is **HELP ABOUTHELP BUILT-IN.**
- The direct path to the topic that describes the **HELP** command is **HELP COMMANDS HELP.**
- The direct path to the topic that describes the **LIST HELP** command is **HELP COMMANDS LIST HELP.**

Therefore, you can display any particular topic in Built-in Help by using a

HELP command that names the downward path (as described in **HELP ABOUTHELP).** to that topic.

(Notice that the **HELP** command given without operands automatically names the Master Index topic.)

Relative References:

A Help topic can be selected by references that are relative to a **current** topic. For more information about **RELATIVE** references, please see **HELP COMMANDS HELP RELATIVE.**

Whenever Z/XDC displays a Built-in Help topic, it sets a **current topic pointer.** Subsequently, a **HELP** command can select the next topic to display via a reference that is relative to the current topic.

The following are examples of relative references:

- **HELP *NEXT** and **HELP *PREVIOUS**
  These display the next or previous topic along the depth-first sequential path.

- **HELP *FORWARD** and **HELP *BACK**
  These display the next or previous topic along a sisters path.

- **HELP *UP** and **HELP *DOWN**
  These display the next or previous topic along a parent-child path.

- **HELP *** and **HELP *REPEAT**
  These both redisplay the current topic.

For more information about the above mentioned paths, see **HELP ABOUTHELP.**

Whenever a Built-in Help topic is displayed at the terminal, the PF key commands are automatically replaced by a special set of **Help Keys** that are more suitable than the normal PF keys for navigating through the Built-in Help structure.

In addition, a **Hints** panel is shown at the bottom of all Help displays so that you will know what the Help Key settings are.

So when you are displaying a Help topic, you will immediately see that most of the PF keys have been set to **HELP** commands that use relative references. For a detailed discussion of the **Help Keys,** see **HELP FULLSCREEN PFKEYS HELPKEYS.**

For more information about relative topic references, please see **HELP COMMANDS HELP RELATIVE.**

Mixing Absolute and Relative Topic References

A Help topic can be selected by a mixture of absolute and relative references. Z/XDC's **HELP** command actually accepts any number of operands each of which can be a relative reference or an absolute topic name.

As the **HELP** command resolves its operands, it temporarily sets a **Current Topic Pointer** to the topic represented by the current operand. Then the next operand is

parsed relative to its prior operand.

So a **HELP** command can use a mix of multiple absolute and relative references to navigate to a desired topic to display. Examples:

- **HELP *UP *FORWARD *DOWN** will display a topic that is a **cousin** to the current topic. (For the topic you are reading right now, in the year of my writing this, that command will display the **HELP MAINTENANCE 2024** topic.)

- **HELP MAINTENANCE 2023 *NEXT *NEXT** will display the 2nd to last maintenance update published in 2023.

Using the LIST HELP Command

A Help topic can be selected by shortcuts from the reports produced by the **LIST HELP** command. This command displays the names of Built-in Help topics at specified levels in the Built-in Help hierarchy. See **HELP COMMANDS LIST HELP** for more information.

When you see a topic that you wish to read, place an **H** in the Shortcut Entry Field to the left, and Z/XDC will display the corresponding topic.

Getting Help for a DBCnnn Message

A Help topic can be selected by shortcuts from any numbered message **(DBCnnnc)** displayed by Z/XDC. Placing an **H** at the underscore (_) or period (.) appearing to the left of a numbered message causes Z/XDC to display the Built-in Help topic that explains that message.

Searching The Help

Individual topics in the live Built-in Help are searchable via the **SCANLOG** command, but the Built-in Help as a whole is not searchable.

To search the Help as a whole, you need to use either the **Online form** or the **PDF form** of the Help. For more information, see:

- **HELP ABOUTHELP ONLINE**
- **HELP ABOUTHELP PDFS**

Help ABouthelp Online

This topic is about the **Online** form of Z/XDC's Help.

Altogether, there are three forms of the Help, all containing the same content:

- **Built-in Help** is built into Z/XDC itself and is directly accessible from a

debugging session via three commands and a shortcut: **HELP, LIST HELP, PRINT HELP** and **H.**

- The **Help PDFs** are available online at **colesoft.com/pdfs.** They can be downloaded and archived on your desktop.

- **Help Online** also is available online at **colesoft.com/help.** It can be accessed outside of a debugging session using just a browser. Also, it is searchable.

To learn about the structure and organization of Z/XDC's Help, please read my parent topic, **HELP ABOUTHELP.**

To learn about the methods for accessing, using and searching the Online form of the Help topics, please continue reading this topic.

Accessing Z/XDC's Help Online

A copy of the Z/XDC Help can be found online at our website at **colesoft.com/help.** This form of the Help is referred to as **Z/XDC Help Online.**

Help Online works entirely in your browser. It has the identical content as the Built-in Help, but unlike the Built-in Help, **Help Online is searchable!**

You do not need any special software to view Help Online. All you need is a browser.

You also don't need a site logon. Any Z/XDC user can peruse the Help online anonymously.

Z/XDC Help Online can be navigated in five distinct ways:

1) A **Topics Index** is displayed in a sidebar at the left. It provides an overview of the topics you're currently viewing, as well as all of its siblings, parents, cousins, and so on.

2) A **searchbar** provides a method for searching topics both by topic name and by matches within the topic bodies.

   - Entering any text will generate a list of topics containing all the individual words of that text occurring either in a topic's title or its body.

   - Entering any text in full quotes will match the entire quote.

   - Examples:

     - Searching for **load modules** will show you topics containing both the word **load** and the word **modules.**

     - Whereas searching for **"load modules"** will only report topics containing the full quote: **"load modules".**

   - You can further restrict your search with the **advanced search options.** These allow you to choose amongst case sensitivity, matching whole

or parts of each phrase and/or using regular expressions.

3) Help Online's search box will also accept **HELP** commands that use absolute operands that form the name of desired topic along with the path to that topic.

   (Help Online does not support the use of relative HELP command operands. We suggest you use the navigation sidebar instead.)

4) You can use the displayed **crosslinks** to jump from their paragraphs directly to other topics containing relevant information.

   When you finish reading a topic, you can return to your prior topic by using the browser's **back** button.

5) The Help pages contain **Prev** and **Next** buttons that you can use to traverse the Help's depth-first sequential path.

Help ABouthelp PDfs

This topic is about the **PDFs** form of Z/XDC's Help.

Altogether, there are three forms of the Help, all containing the same content:

- **Built-in Help** is built into Z/XDC itself and is directly accessible from a debugging session via three commands and a shortcut: **HELP, LIST HELP, PRINT HELP** and **H.**

- **Help Online** is available online at **colesoft.com/help.** It can be accessed outside of a debugging session using just a browser. Also, it is searchable.

- The **Help PDFs** also are available online at **colesoft.com/pdfs.** They can be downloaded and archived on your desktop.

To learn about the structure and organization of Z/XDC's Help, please read my parent topic, **HELP ABOUTHELP.**

To learn about the available Z/XDC documentation in PDF form, please continue reading this topic.

What's Available

All of the manuals documenting Z/XDC can be found at **colesoft.com/pdfs.** They are all in PDF format, and they all can be downloaded as needed by Z/XDC customers.

Broadly speaking, the manuals fall into two categories:
- Those that are built upon the content of Z/XDC's Built-in Help
- Those that are not.

PDFs Based on the Built-in Help

The following PDFs are direct extractions of various parts of this Built-in Help.

- **Maintenance and New Functionality Guide:** This document is intended to let customers stay up to date whenever a new update is installed against Z/XDC. The Guide thoroughly documents what is new and changed. It is built from **HELP MAINTENANCE** and its subtopics.

- **Commands Reference:** This manual is a comprehensive reference documenting all Z/XDC commands. It provides:
  - Syntax details
  - Usage information
  - Usage examples

  It is built from:
  - **HELP COMMANDS** and its subtopics.
  - As well as **HELP SHORTCUTCMDS** and its subtopics.

- **Messages Reference:** This manual documents all numbered **DBCnnnc** messages. The discussions tend to be exhaustive. It is built from **HELP MESSAGES** and its subtopics.

- **Users Guide:** This manual contains numerous tutorials about various aspects of using Z/XDC to debug various kinds of programs running in various z/OS execution environments. It is built from everything else.

Additional Manuals

The following manuals provide additional information about Z/XDC

- **z/XDC Primer:** This is a task-oriented, how-to guide for using Z/XDC.

- **z/XDC Quick Reference:** This gives brief descriptions and syntax diagrams for all Z/XDC commands.

- **z/XDC Installation Guide:** This provides detailed information about installing Z/XDC onto your system. It is intended for SysProgs who are responsible for installing and maintaining mainframe software.

As noted above, all of these manuals are in PDF format and can be downloaded from **colesoft.com/pdfs.**

Training Videos

In addition, a growing set of Z/XDC Training Videos can be found on YouTube at **www.youtube.com/user/ColeSoftware.**

Help ABouthelp Conventions

In general the following conventions hold for the syntactical descriptions shown in the various HELP displays.

In the various descriptions of Z/XDC commands and their operands, uppercase words are keywords and should be typed as is. Lowercase words and phrases are variables. Appropriate values should be substituted for them.

Double quotes ("), underscores (_), and ellipses (...) are "meta-characters". They are used only to aid in descriptions of syntax. They should never be used when actually typing Z/XDC commands.

Double quotes (") are used to frame sample commands or keywords as well as various other objects.

Underscores (_) are used to connect together the words appearing in a lowercase phrase. This emphasizes that the several words are descriptive of and represent only a single variable rather than several variables.

Ellipses (...) are used to indicate either repetitive syntax or omitted syntax.

Help ABouthelp PRinting

- You cau use the **PRINT HELP** command to print out any collection of Built-in Help topics you like. For detailed information, see HELP COMMANDS PRINT HELP.

- By default, the selected topics will be written to JES spool in sysout class X with HOLD=YES. But the default can be changed using the **SET PRINT** command.

- Any part of, or all of Z/XDC's Built-in Help can be printed via the **PRINT HELP** command. Individual topics or entire branches of topics can be selected for printing. The output is paginated and written to whatever file is allocated to ddname **XDCPRINT.** If XDCPRINT is not already allocated, then it is automatically allocated to a sysout output class.

  XDCPRINT can be preallocated to any output device or dataset supported by QSAM; however, if it is not preallocated, then Z/XDC will dynamically allocate it only to a sysout class on spool.

- XDCPRINT can be preallocated either via an //XDCPRINT DD card or via TSO's ALLOCATE command (assuming that Z/XDC is running within TSO). ALLOCATE either can be issued prior to the start of the debugging session, or it can be issued during the debugging session via Z/XDC's **TSO** command.

  The maximum number of characters that Z/XDC writes to the XDCPRINT file per output record is 80 (including an ASA carriage control character). XDCPRINT supports any reasonable DCB attributes for the XDCPRINT file so long as it accommodates an 80-byte record length:

  - The record format (RECFM) may be fixed, variable, or undefined. The records may be blocked or unblocked. Z/XDC will always use ASA carriage control.

  - Any legal LRECL and BLKSIZE values may be set, but if a preset LRECL is not

large enough to permit 80 characters per record, then truncation will occur.

When Z/XDC opens XDCPRINT, it may discover that one, some, or all of the file's DCB attributes are not set. In this case, Z/XDC will set default values that (a) are consistent with those values that are set, and (b) come as close as is reasonable to Z/XDC's preferred defaults: RECFM=VBA, LRECL=84, and BLKSIZE=6233.

Z/XDC has three commands that relate to printing Built-in Help topics. They are PRINT HELP, LIST PRINT, and SET PRINT:

PRINT HELP ...
This command is used to select the Built-in Help topics to be printed and to write them to the XDCPRINT output file. PRINT HELP allocates the XDCPRINT file and OPENs it, when necessary.

LIST PRINT
This command displays a report that shows both certain characteristics of the XDCPRINT file and its current status: OPEN'd or CLOSE'd, allocated, not allocated, etc.

SET PRINT ...
This command can be used both to set certain characteristics of the XDCPRINT output file and to change its status. The characteristics that can be set are:

- Sysout class.
- Output hold status.
- Lines to print per page.

The possible status changes that can be performed are:

- **SET PRINT CLOSE:** Appends a Table of Contents to the output, closes it, and deallocates it (but only if Z/XDC allocated it in the first place).

- **SET PRINT DELETE:** Closes, deallocates, and attempts to delete the XDCPRINT file. The deallocation will occur regardless of whether or not Z/XDC allocated the file. The deletion can fail for any of several reasons including, the allocation to XDCPRINT was shared, not exclusive.

Help COmmands

(For descriptions of all z/XDC® commands, line commands, and command files, see the z/XDC® Commands manual.)

Help CMddialogs

Debugging can be quite a complex process, and Z/XDC provides a vast array of tools and procedures to help you in your efforts. But of course, some of the commands that you might use are pretty complex, and if you don't use them frequently, they can be hard to remember.

To help deal with this problem, Z/XDC provides a facility we call **Command**

Dialogs. These are easily invoked panels that provide both commentary and fill-in fields that make it easier to do such things as load maps, search storage and many other things.

The dialogs contain clearly captioned fill-in fields that allow you to tell a command what you want done. The commentary attempts both to explain what your choices are and to illuminate some of the considerations and consequences pertaining to those choices.

One important thing: Command Dialogs, in order to be useful, need to present a pretty **large amount** of information. Therefore, they don't really work very well on display screens with small row-by-column geometries. So even though you can scroll up, down, left or right, we nevertheless believe that you will find the dialogs to be the most useful when used on terminals with large display geometries. For comprehensive information about setting up a large geometry display, see **HELP FULLSCREEN TERMINALS GEOMETRIES.**

At the moment, we have only four Command Dialogs available. They are for the **DMAP, FIND, MAP** and **USING** commands. But we will be writing more Dialogs and making them available via maintenance and new releases. So please stay tuned...

There also is a **Command Dialogs Index** panel that lists all of the available Dialogs and allows you to select which ones you want to work with. To display the Index, simply enter a lone **?** (question mark) on the command line and then press ENTER.

Help SHortcutcmds

Most lines displayed at the terminal have underscores (_) or periods (.) showing at the left side of the screen. These are **Shortcut Entry Fields,** and any of several 1- or 2-character shortcuts can be entered there. Different display lines accept different sets of shortcuts.

- Shortcut Entry Fields designated by **periods** accept only the **H** and **?** shortcuts.
- Shortcut Entry Fields designated by **underscores** accept a wider selection of shortcuts from those listed below.

Below is a list of all shortcuts supported by Z/XDC.

- Different Shortcut Entry Fields will accept different subsets of these shortcuts.
- No single Shortcut Entry Field will accept all them.
- For any given Shortcut Entry Field, you can use a **?** to show a list of all shortcuts that particular field accepts.

Shortcuts Index

The following detailed information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

ASSOCADDR - Most display lines generated by Z/XDC are related to some object or location in storage. Thus, such lines have an **associated**

address assigned to them. The presence of this associated address strongly affects which shortcuts will be accepted for a given shortcut input field.

? - **("more info")** Each Shortcut Entry Field accepts only a **subset** of the following shortcuts. To see a list of the particular shortcuts a particular field accepts, use the **question mark.**

A - Issues an **AT** command.
CU - Issues a **SET CURRENT** command (dump/XDC only).
D - Issues a **DISPLAY** command.
E - Issues an **EWHERE** command.
F - Issues a **FORMAT** command.
H - Issues a **HELP** command.
J - ("jump") Issues a **ZAP PSW address** command.
K - Issues a **HOOK** command.
L - Issues any of several **LIST** commands.
LL - Issues a **LIST LOCATION** command.
LO - Issues a **LIST OPERANDS** command.
M - Issues a **MAP** command.
N - Issues a **NOTE** command.
O - Issues an **OFF** command.
P - **Purges** a note.
Q - Issues a **SET QUALIFIER** command.
S - Issues a **SWITCH** command or performs a **selection.**
SH - Issues a **SHOW** command.
T - Issues a **TRAP** command.
W - Issues a **WHERE** command.
X - Issues a **SET BREAKPOINT TOGGLE** command.
Z - Issues a **ZAP** command.

c/XDC Related Shortcuts

The following shortcuts can be used only on c/XDC's **LIST VARIABLES** command:

***** - The listed string variable is redisplayed interpreted as **EBCDIC.**

| - The listed string variable is redisplayed interpreted as **ASCIIC.**

**** - ("string") The listed string variable is redisplayed interpreted as a **null delimited character string.**

/ - ("array") The listed string variable is redisplayed interpreted as a **fixed length array** of characters.

Help SHortcutcmds ASSocaddr

Most display lines generated by Z/XDC are related to some object or location in storage. For example:

- In a **LIST TASKS** display, each line describing a TCB is related to the address of that TCB.
- Each line in a **LIST RBS** display is related to the location pointed to by the described RB's resume PSW.
- Each line in a **LIST PGMS** display is related to the described load module's entry address.
- Etc.

Accordingly, for those display lines that relate to storage locations, Z/XDC assigns an **associated address.** The presence of this associated address strongly affects which shortcuts will be accepted for a given Shortcut Input Field (_). Specifically:

- Any line that has an associated address will accept **D F M N Q** and **SH** shortcuts. This includes not only lines that refer to storage, but also lines that actually display storage as well.

- Lines that actually display storage will, in addition, accept the storage altering shortcuts **(A J K O T X** and **Z)** as well as the **L** and **W** shortcuts. (Lines that only **refer** to storage but do not display storage will not accept these storage altering shortcuts.)

- The remaining shortcuts **(H P S** and **?)** can be used or not depending upon factors other than associated address.

- The **?** shortcut can be used in **any** Shortcut Entry Field to display a list of the specific shortcuts permitted by that field.

The following table lists the associated addresses assigned to many of the displays produced by various Z/XDC shortcuts.

COMMANDS	ASSOCIATED ADDRESSES
DISPLAY	- The starting address of each storage display line.
FIND	- The starting address of each storage display line.
FORMAT	- The starting address of each storage display line.
GETMAIN	- The location of the obtained area of storage.
LIST BREAKPOINTS	- Breakpoint locations.
LIST CRn	- When a control register contains a pointer to a location in real or virtual storage (page table, trace table, linkage stack entry, etc.), the display message will have an associated address pointing to that object.
LIST EPSW	- The location pointed to by the error level PSW (i.e. where the user ABEND occurred).
LIST EQUATES	- The locations labeled by the equates.
LIST ESTAES	- The starting locations of the ABEND exit recovery routines pointed to by the displayed SCBs.
LIST FIXED	- The location being displayed.
LIST FLOAT	- The location being displayed.
LIST VARIABLES	- The location of the High Level Language Variable.
LIST HOOKS	- The location of the listed hook.
LIST LKEDMAP	- The locations of the listed csects and entry names.
LIST LSTACK	- Various locations depending upon the information being displayed.

LIST MAPS - The locations that are formatted by the listed maps.
LIST MEMORYOBJECTS - The locations of the displayed memory objects.
LIST MSGS - The location of the user ABEND or breakpoint that caused Z/XDC to receive control.
LIST NOTES - The locations to which the listed notes refer.
LIST PGMS - The entry points of the listed load modules.
LIST PSW - The location pointed to by the retry level PSW (i.e. where execution will resume when you issue the **GO** or **TRACE** command).
LIST QUALIFIER - The qualified location.
LIST RBS - The listed Request Block's resume execution locations (determined by the next instruction address field of the PSWs that are stored in the Request Blocks).
LIST RSA - The locations of the listed register save areas.
LIST SSCT - For display lines that report on subsystem function routines, the associated address is said routine's entry point. For display lines that do not include subsystem routine information, the associated address is the location of the subsystem's SSCT.
LIST SUBPOOLS - The locations of the listed blocks of storage.
LIST TASKS - The locations of the listed task control blocks.
LIST TIOT - The locations of the individual TIOT DD entries.
LIST VSTACK - The locations of the listed variable pool, or the execution address.
LOAD - The entry point of the LOAD'd module.
MAP - The location(s) of the MAP'd object(s).
SET QUALIFIER - The qualified location.
SHOW - The starting address of each storage display line.
WHERE - The starting address of each storage display line.

Help SHortcutcmds ?

The **?** ("more info please") shortcut has two different uses depending upon whether or not the current display is of a **HELP** topic:

- Outside of Built-in Help, the **?** shortcut can be used in **any** Shortcut Entry Field to display a list of all shortcuts that will be accepted by that specific field. The list is displayed via a numbered message **(DBC732I)** appearing on the top window's command line.

- Within Built-in Help, the Shortcut Entry Fields are used (via **H** shortcuts) to follow crosslinks between the current HELP topic and other HELP topics containing related information. If a **?** shortcut is used instead, then the crosslinking **HELP** command is displayed instead of the target HELP topic. This allows you to see what the crosslinks are without having to display the target topic. See **HELP ABOUTHELP** for more information.

Help SHortcutcmds A

A is a shortcut for the **AT** command. It is used to set **persistent** breakpoints at displayed locations in the Primary Address Space.

T is a shortcut for the **TRAP** command. It is used to set **transient** breakpoints.

For a discussion of **persistent** vs. **transient** breakpoints, see **HELP BREAKPOINTS TRANSIENT.**

Except for the type of breakpoint set, the **A** and **T** shortcuts behave identically.

They can be typed into the Shortcut Entry Fields (_) of display lines generated mainly by the various storage display commands:

- **DISPLAY**
- **EWHERE**
- **FIND**
- **FORMAT**
- **LIST LKEDMAP**
- **SHOW**
- **WHERE**

Generally, other lines do not accept **A** and **T.**

The displayed storage must meet all of the following restrictions:

- The storage can be located in any accessible and permitted address space. It may not be located in a data space or in real storage.
- The location must be halfword aligned.
- The location must be store accessible to Z/XDC. This means that:
  - It must be allocated (i.e. it must have been previously obtained via a GETMAIN or STORAGE request).
  - If Z/XDC is running non-authorized, then the storage's access key must either be key 9 or must match Z/XDC's execution key (usually key 8).
  - System Security must permit the store-access.
- If Z/XDC is running authorized, then all of the above restrictions still apply, except that Z/XDC can place a breakpoint into storage without regard to:
  - The storage's access key.
  - Whether or not the storage is store protected.

  Z/XDC cannot place a breakpoint into the first 512 bytes of address space storage.

Note the Following Features

- The first byte at the location where a breakpoint is to be set does not have to contain a valid machine instruction opcode. Z/XDC will set the breakpoint anyway.
- Breakpoints may already exist at the target address. Z/XDC supports setting multiple breakpoints at any given location. (This can be useful if the multiple breakpoints have differing characteristics.)

- Z/XDC supports setting breakpoints on instructions that are executed via **EX** and **EXRL** machine instruction.

- If Z/XDC is running authorized, then it is possible to use the **A** and **T** shortcuts to set breakpoints into any valid location in the PLPA, FLPA, MLPA, and System Nucleus, **even in** the store protected portions of these areas. **GREAT CARE MUST BE EXERCISED WHEN USING THIS FEATURE! INCORRECT USAGE CAN EASILY CAUSE YOUR SYSTEM TO CRASH!**

- You may type **A** and **T** shortcuts into as many Shortcut Entry Fields (_) as you choose before pressing the ENTER key. When you do press ENTER, one breakpoint will be set for each **A** and **T** shortcut that Z/XDC accepts. All of the breakpoints thus created will be assigned the same family identifier. For a discussion of breakpoint families, see **HELP BREAKPOINTS FAMILIES.**

Help SHortcutcmds Cu

CU is a shortcut for the **SET CURRENT** command. It has meaning only when using dump/XDC to troubleshoot a dump.

CU nominates an execution thread in a dump as being the **current** thread. It can designate either a task and RB combination or an SRB.

Doing this affects numerous Z/XDC commands that by default target their displays or other actions against a **current** task or SRB.

You can also use **CU** to reset the CAS/CXU to the dump's default (if any).

CU is available only when using dump/XDC. It has no meaning outside of dumps.

For more information about dump/XDC's management of notions of what's current and what's not, see **HELP DUMPS EXECUTIONTHREAD.**

You may change the current execution thread to whatever thread you like, and you may make the change as often as you want. The effect is immediate and is remembered by dump/XDC so that it persists across dump/XDC sessions. The change affects all commands (going forward) whose default action is to target the current execution thread. Examples:

- **LIST TASKS:** By default, this displays the task tree in the designated current address space.
- **LIST RBS:** By default, this displays the request blocks queued to the current task.
- **LIST RWREGS:** By default, this displays the 8-byte wide general registers associated with the current request block.

All of these commands continue, of course, to accept operands that can redirect their attention to whatever execution thread catches your interest.

dump/XDC uses two concepts to identify a current execution thread:
- **CAS:** This designates a "Current Address Space".
- **CXU:** This designates a "Current eXecution Unit". This may be either a request block linked to a task, or an SRB.

The CAS and CXU can be set either by setting certain IPCS literals, or by this CU shortcut, or by a SET CURRENT command. For further details, see **HELP DUMPS EXECUTIONTHREAD.**

The **CU** shortcut can be used on lines that display information about any of the following:

- **ASIDs** The designated address space is made the CAS, and its jobstep task and newest RB are made the CXU.
- **Tasks** The designated task's newest RB is made the CXU, and the containing address space is made the CAS.
- **RBs** The designated request block is made the CXU, and the containing address space is made the CAS.
- **SSRBs** The designated SRB is made the CXU, and the owning address space is made the CAS.

- **CPUs** When processing a stand-alone dump, the dump may contain information about the execution state of every CPU within the system, and the **LIST DXDC CPUS** command displays that information. The CU shortcut can be used to designate a CPU as being current, but only if the execution thread's home address space is present in the dump.

In addition, the **CU** shortcut can be used on one of the **LIST DXDC** report lines in the **Current Information** section of the report to revert the current execution thread to dump/XDC's initial setting.

The following is a partial list of commands that produce displays that accept **CU** shortcuts where appropriate:

LIST DXDC

LIST RBS

LIST TASKS

LIST SSRBS

Help SHortcutcmds D

D is a shortcut for the **DISPLAY** command. It is used to display storage using a raw hex-text (dump like) format.

F is a shortcut for the **FORMAT** command. It is used to display storage using Z/XDC's symbolic storage formatter that disassembles machine instructions, displays labels,

and breaks out control blocks field by field.

In all other aspects, the **D** and **F** shortcut commands behave identically. For further information see **HELP SHORTCUTCMDS F.**

Help SHortcutcmds E

E is a shortcut for the **EWHERE** command. It generates a storage display that shows program code at the user program's error level ABEND address.

E can be used on any display line that shows storage located in the Primary Address Space. It will be successful, however, only when it is placed at or prior to (but still near) the location to which the **error level** PSW points.

E will attempt to build a new display that meets two criteria:
- The line on which the **E** was placed is repositioned to the top of the display.
- The display still includes the error level's execution address (i.e. the location to which the error level PSW points).

If the rebuilt display fails to include the error level execution address, it is discarded, and a replacement **EWHERE** is processed with the old **EWHERE** positioning retained.

Note:

- Generally, the **WHERE** and **EWHERE** commands will produce similar displays **except** when the retry level and error level environments are **different.** Then the **WHERE** command will show the **retry level** resume address, while the **EWHERE** command will show the **error level** ABEND address. For more information, see **HELP EXECUTIONLEVELS.**

Example:

Suppose that Z/XDC's **WHERE, EWHERE** or **TRACE** command had been used to produce the following display:

```
_ 003210 -- ARTEST.X2+0
_    .+0 47F0 F01A      X2       BC    B'1111',1A(,R15)
_    .+4          15              21                **.*
_    .+5 E0005MID E7F14040 404040                   **X1     *
_    .+C          F0F361F3 F061F9F4 40              **03/30/94 *
_   .+15          F1F34BF1 F6                       **13.16*
_   .+1A 90EC D00C      E0005ZID STM   R14,R12,C(R13)
_   .+1E 181D                    LR    R1,R13
_   .+20 41D0 F034               LA    R13,34(,R15)
_   .+24 50D0 1008     >         ST    R13,8(,R1)
_   .+28 5010 D004               ST    R1,4(,R13)
_   .+2C 5810 1018               L     R1,18(,R1)
```

```
_   .+30 47F0 D048              B     B'1111',48(,R13)
_   .+34 E0005SVA 00000000 00000000 00000000 00000000 *................*
```

In the above display:
- the user program's error level ABEND address is the hilighted line at location **.+24.**
- If an **E** shortcut is placed, for example, on the line at location **.+1A,** then a new display will be generated that starts at that address.
- On the other hand, if a **E** were placed on a line following the error level ABEND address, such as at location **.+30,** then the new display that would be generated would not be adjusted at all.
- It would still start at location **.+0.**
- This is because displays generated by **EWHERE** commands **must** include the user program's error level ABEND address.
- See **HELP COMMANDS EWHERE** for more information.

Help SHortcutcmds F

F is a shortcut for the **FORMAT** command. It displays storage using Z/XDC's symbolic storage formatter that disassembles machine instructions, displays labels, and breaks out control blocks field by field.

D is a shortcut for the **DISPLAY** command. It is used to display storage using a raw hex-text (dump like) format.

In all other aspects, the **F** and **D** shortcuts behave identically.

The **F** and **D** shortcuts can be used to display the storage that is being referred to by any display line that has an associated address.

If the line receiving the **F** or **D** shortcut already is displaying storage, then the same storage is simply reformatted according to the shortcut used **(F** or **D)** and repositioned according to the line on which the **F** or **D** shortcut was placed.

For more information about associated addresses, as well as an extensive list of Z/XDC's displays and their associated addresses, see **HELP SHORTCUTCMDS ASSOCADDR.**

Help SHortcutcmds H

H is a shortcut for the **HELP** command. It displays topics from Z/XDC's Built-in Help Database. It can be used in three situations:

1) **H** can be issued against any numbered Z/XDC message **(DBCnnn)** to obtain the specific Built-in Help topic for that message. Numbered messages always start with the letters **DBC** followed by a 3-digit or 4-digit number. Example:

```
    XDC ===> junkcmd
    JUNKCMD
          *
```

h DBC007E COMMAND NOT RECOGNIZED
Here, the **h** shortcut displays the **HELP MESSAGES DBC007** topic.

2) Either **H** or **S** can be issued against the reports produced by the **LIST HELP** command to display the Built-in Help topics listed by the report. Example:
XDC ===> list help attentions
...
_ Help ATtentions
_ . INXdc
h . INUserpgm
In this case, the **H** shortcut displays the **HELP ATTENTIONS INUSERPGM** topic.

3) And finally, either **H** or **S** can be used within **Built-in Help** to follow crosslinks to other topics. Example? Ok, just put an **H** into the **_** located at the left of **this** paragraph, and press ENTER.

(Press the **RETRIEVE** key **[F12]** twice to get back.)

Help SHortcutcmds J

J is a shortcut that effects a **jump** of the retry level execution resume address. In effect, it does a **ZAP PSW address** command.

J does not cause the user program to resume. It simply changes the resume PSW such that when you eventually issue **TRACE** or **GO,** user program execution will resume to the instruction you issued the **J** against.

The **J** command can be issued against any screen line that displays storage located within the Primary Address Space.

Help SHortcutcmds K

K is a shortcut for the **HOOK** command. It attempts to place a hook at the displayed storage location. It can be typed into the Shortcut Entry Fields of display lines generated mainly by the various storage display commands:

- **DISPLAY**
- **EWHERE**
- **FIND**
- **FORMAT**
- **LIST LKEDMAP**
- **SHOW**
- **WHERE**

Help SHortcutcmds L

L is a shortcut for the **LIST** command. It can be used to issue any of several Z/XDC LIST subcommands depending upon the display line against which it is issued.

The following is a partial list of displays that accept the **L** shortcut, and the subsequent displays that result.

TARGET DISPLAY **COMMAND ISSUED BY THE L SHORTCUT**

LIST DXDC - There are 2 cases where the **L** shortcut can be used on the **L DXDC** summary display, depending the attributes of the current execution unit:

- **LIST LOCKS**
  If the current execution unit is holding any locks, then the **L** shortcut can be used to issue a **LIST LOCKS** command on the line that indicates that the **CXU** is holding locks.

- **LIST XMS**
  If the current execution unit is in a cross-memory mode (HASID<>PASID<>SASID), then the **L** shortcut can be used on the appropriate report line to issue a **LIST XMS** command.

LIST FIXED - **LIST FLOAT**
The same location is redisplayed, but formatted as floating point numbers, all three formats: HFP, BFP and DFP.

LIST FLOAT - **LIST FIXED**
The same location is redisplayed, but formatted as 1-byte, 2-byte, 3-byte and 4-byte fixed point numbers.

LIST HELP - **LIST HELP various operands**
A new LIST HELP report is displayed that shows either a **lower** or **higher** topic level than that which is currently displayed.

LIST LKEDMAP - **LIST LKEDMAP FULL**
The linkedit map is redisplayed but with **FULL** specified. In other words, the map is reshown but with the maximum amount of information available.

LIST LSTACK - **LIST LSTACK**
The linkage stack display is toggled back and forth between a summary display and a detailed display.

LIST PGMS - **LIST LKEDMAP modulename CSECTS**
A linkedit map is displayed showing all of the csects (but not the entry points) comprising the selected load module.

LIST QUALIFIER - **LIST LKEDMAP modulename CSECTS**

A linkedit map is displayed showing all of the csects (but not the entry points) comprising the qualified load module.

LIST RBS - **LIST RWREGS;LIST AREGS;LIST FREGS;LIST CRWREGS**
A display is produced of all registers in current use by the program running under the selected Request Block. All register types are shown: 64-bit general, 32-bit access, 64-bit floating point and 64-bit control.

LIST SUBPOOLS - **LIST SUBPOOLS nnn DETAILS**
When used on a **TOTALS** line, a DETAILS display is produced of the totaled subpool.

LIST TASKS - **LIST RBS TCB#nnn**
A display is produced of all of the Request Blocks running under the selected task.

LIST VARIABLES - **LIST VARIABLES various operands**
Here, The **L** shortcut allows you either to **drill down** into a structure or to **climb back out.** Its behavior differs, though, according to whether the LIST VARIABLES display is in the Primary Window or a Watch Window:

- For **Watch Windows,** the display is either expanded or or collapsed in place.

- For **Primary Windows,** on drill down, a new display is produced showing just the variable expansion. On climb out, a new display is produced showing just the collapse of the variable.

LIST VSTACK - **LIST VARIABLES various operands**
A display is produced of all of the C Language variables and structures within the selected variable pool.

MAP - **LIST LKEDMAP modulename CSECTS**
A linkedit map is displayed showing all of the csects (but not the entry points) comprising the mapped load module.

SET QUALIFIER - **LIST LKEDMAP modulename CSECTS**
A linkedit map is displayed showing all of the csects (but not the entry points) comprising the qualified load module.

Load module Storage - **LIST LKEDMAP +0 CSECTS**
A linkedit map is displayed showing all of the csects (but not the entry points) comprising the load module.

C Variable Storage - **LIST VARIABLES various operands**
The location is displayed formatted as a C Language variable within it’s containing structure.

Data area storage - **LIST FIXED +0**
The location is displayed formatted as 1-byte, 2-byte, 3-byte and 4-byte fixed point numbers. The 2-byte and 4-byte numbers are displayed as signed. The 1-byte and 3-byte numbers are displayed as unsigned.

Help SHortcutcmds LL

LL is a shortcut for the **LIST LOCATION** command.

When a message refers to a location in storage, **LL** shows Z/XDC's analysis of where that location is:

- What load module it is within
- What csect it is within
- What equate it is within
- What dsect it is within

If the loction is within one or more objects that Z/XDC knows about, **LL** will show what those objects are.

For more informatioon, see **HELP COMMANDS LIST LOCATION.**

Help SHortcutcmds LO

LO is a shortcut for the **LIST OPERANDS** command. It can be used to display the storage referenced by the operands of **any** machine instruction.

Z/XDC's knowledge of z/System machine instructions is comprehensive. No matter what instruction you target with the **LO** command, if that instruction references storage in any way, this command will display all of the references.

WARNING! When decoding a machine instruction's **register** based storage references, Z/XDC uses whatever the retry level register's **current** value is. So if those current values will be changed by the time execution actually reaches that instruction, the actual storage referenced will be **different** from what this command displays. So clearly, this command is pretty useless unless it is used on instructions that are logically/conceptually near the current execution location.

Help SHortcutcmds M

M is a shortcut for the **MAP** command.It can be used to cause Z/XDC to read a module map and, possibly, a csect map of a load module.

The address associated with the line receiving the **M** command has to be located within a load module in order for the **M** command to work correctly. For more information about associated addresses, as well as an extensive list of Z/XDC's displays and their associated addresses, see **HELP SHORTCUTCMDS ASSOCADDR.**

Help SHortcutcmds N

N is a shortcut for the **NOTE** command. It can be issued against any display line that has an associated address. It copies the target line to Z/XDC's list of noted addresses.

The notes list can be displayed by the **LIST NOTES** command.

For more information about associated addresses, as well as an extensive list of Z/XDC's displays and their associated addresses, see **HELP SHORTCUTCMDS ASSOCADDR.**

Help SHortcutcmds O

O is a shortcut for the **OFF** command. It removes breakpoints. It can be used in the following displays:

- On any line that displays storage containing one or more **breakpoints** in the first byte displayed:
  - If a mix of breakpoints are present, then all but **ATX breakpoints** (if any) are removed.
  - If only one or more ATX breakpoints are present, then all of the ATX breakpoints are removed.
- On any line that displays storage containing a **hook.** The hook is removed.
- On lines produced by the **LIST BREAKPOINTS** command. In this case, only the specifically selected breakpoints are removed.
- On lines produced by the **LIST HOOKS** command. The selected hook is removed.

Help SHortcutcmds P

P is a shortcut for **purging** entries from a list. It can be used in the following displays:

LIST NOTES - P removes individual note entries from the display.
RETRIEVE LIST - P removes individual command strings from the stack of retrievable commands.

Help SHortcutcmds Q

Q is a shortcut for the **SET QUALIFIER** command. It can be used to set a default load module name and csect name to be used in address expressions that start with a leading dot **(.).**

Also, when the **WHERE, FORMAT** and similar commands are used to display code within the qualified csect, offset displays are shown as **.+offset** (instead of just plain **+offset).**

See **HELP ADDRESSING DOTPLUS** for more information about using default qualifiers.

Help SHortcutcmds S

The **S (Select** or **SWITCH)** command has several uses depending upon the display against which it is issued:

TARGET DISPLAY	RESULT
LIST TASKS	- In this case **S** means **SWITCH.**
	In multitasking debugging, if Z/XDC has received control simultaneously under multiple tasks, then the **S** command, when used against a **LIST TASKS** display, causes Z/XDC to suspend itself in the current task and switch to another copy of Z/XDC that, until now, has been suspended in the target task. For more information, see **HELP MULTITASK.**
	Eligible tasks are those tasks that are flagged with **PENDING XDC** in the **LIST TASKS** display.
LIST HELP	- Either **S** or **H** can be issued against the reports produced by the **LIST HELP** command to display the Built-in Help topics listed by the report.
Built-in HELP	- When a paragraph of Built-in HELP information (such as this one) has a Shortcut Entry Field, that means that paragraph contains a crosslink to another Built-in Help topic containing related information. Use the **S** or **H** shortcut to follow that crosslink. (Use the **?** shortcut to display that crosslink without following it.) For more information, see **HELP ABOUTHELP.** (You can type an **S** (or **H)** at the left of this paragraph to do so.)
Profile Menus	- When displaying menus (panels) in the **Profile Menuing System,** when one menu (such as the top menu) has submenus linked to it, you can use the **S** shortcut to display the submenu. For more information about the **Profile Menuing System,** see **HELP PROFILES.**
Retrieval Stack	- When you use the **RETRIEVE LIST** command, you will be shown a list

of commands that you have previously issued. If you wish to reissue one of those commands, you can use an **S** shortcut to select the one you want. It will be copied to the command line where you can edit and then press ENTER to issue the command. For more information, see **HELP COMMANDS RETRIEVE LIST.**

Job Selection - When you invoke **cdf/XDC,** the first thing you will see is a **Job Selection Menu.** You can use the **S** shortcut to select which job you wish to debug. For more information, see **HELP XDCSRVER CDF.**

Help SHortcutcmds SH

SH is a shortcut for the **SHOW** command. It displays a **single** line of storage at the location associated with the message. For more information, see **HELP COMMANDS SHOW.**

Help SHortcutcmds T

T is a shortcut for the **TRAP** command. It is used to set transient breakpoints at displayed locations in the Primary Address Space.

A is a shortcut for the **AT** command. It is used to set persistent breakpoints.

In all other aspects, the **A** and **T** shortcut commands behave identically. For further information see **HELP SHORTCUTCMDS A.**

Help SHortcutcmds W

W is a shortcut for the **WHERE** command. It generates a storage display that shows the user program’s retry level resume address.

W can be used on any display line that shows storage located in the Primary Address Space. It will be successful, however, only when it is placed at or prior to (but still near) the location to which the **retry level** PSW points.

W will attempt to build a new display that meets two criteria:
- The line on which the **W** was placed is repositioned to the top of the display.
- The display still includes the retry level’s resume address (i.e. the location to which the retry level PSW points).

If the rebuilt display fails to include the retry level resume address, it is discarded, and a replacement **WHERE** is processed with the old **WHERE** positioning retained.

Notes:

- When **SET TRACE ROLL** is in effect, the **TRACE** command internally uses the **WHERE** command to produce its displays such that the resume address (which, of course, advances during tracing) has the appearance of rolling down the display screen.

- Generally, the **WHERE** and **EWHERE** commands will produce similar displays **except** when the retry level and error level environments are different. Then the **WHERE** command will show the **retry level** resume address, while the **EWHERE** command will show the **error level** ABEND address. For more information, see **HELP EXECUTIONLEVELS ERRORLVL.**

Example:

Suppose that Z/XDC's **WHERE, EWHERE** or **TRACE** command had been used to produce the following display:

```
_ 003210 -- ARTEST.X2+0
_    .+0 47F0 F01A      X2       BC    B'1111',1A(,R15)
_    .+4          15              21                **.*
_    .+5 E0005MID E7F14040 404040                   **X1     *
_    .+C          F0F361F3 F061F9F4 40              **03/30/94 *
_   .+15          F1F34BF1 F6                       **13.16*
_   .+1A 90EC D00C      E0005ZID STM   R14,R12,C(R13)
_   .+1E 181D                    LR    R1,R13
_   .+20 41D0 F034               LA    R13,34(,R15)
_   .+24 50D0 1008     >         ST    R13,8(,R1)
_   .+28 5010 D004               ST    R1,4(,R13)
_   .+2C 5810 1018               L     R1,18(,R1)
_   .+30 47F0 D048               B     B'1111',48(,R13)
_   .+34 E0005SVA 00000000 00000000 00000000 00000000 *................*
```

In the above display:
- the user program's retry level resume address is the hilighted line at location **.+24.**
- If a **W** shortcut is placed, for example, on the line at location **.+1A,** then a new display will be generated that starts at that address.
- On the other hand, if a **W** were placed on a line following the retry level resume address, such as at location **.+30,** then the new display that would be generated would not be adjusted at all.
- It would still start at location **.+0.**
- This is because displays generated by **WHERE** commands **must** include the user program's retry level resume address.
- See **HELP COMMANDS WHERE** for more information.

Help SHortcutcmds X

X is a shortcut for the **SET BREAKPOINT TOGGLE** command. It causes breakpoints at the displayed storage location to be toggled on or off.

All breakpoints at that location are toggled: Enabled breakpoints are disabled, and disabled breakpoints are enabled.

If the location contains a mix of breakpoints that include **ATX breakpoints,** then then all but the ATX breakpoints (if any) are toggled.

If the location contains **only ATX** breakpoints, then all of the **ATX** breakpoints are toggled.

The **X** shortcut can be used only at storage locations for which enabled or disabled breakpoints are defined. It will fail at locations for which no breakpoints are defined. For more information, see **HELP COMMANDS SET BREAKPOINTS.**

Help SHortcutcmds Z

Z is a shortcut for the **ZAP** command. It can be used to alter the contents of:
- Data areas
- Machine instructions
- Virtual storage in general
- Retry level general registers
- Retry level access registers
- Floating point registers
- Vector registers

Z can be issued against displays produced by the following commands:

DISPLAY
EWHERE
FIND
FORMAT
LIST AREGS
LIST ARn
LIST FIXED
LIST FLOAT
LIST FREGS
LIST VARIABLES
LIST REGS, LIST RHREGS, LIST RWREGS
LIST Rn, LIST RHn, LIST RWn
LIST VREGS
SHOW
WHERE

And probably others...

Using the Z Shortcut

To use the **Z** shortcut command, simply type **Z** into the Shortcut Entry Fields of all lines that you want to zap, then overtype new data directly into the writable fields of the displayed lines, then press ENTER.

If you type data without also typing **Z** at the left, then your data will be ignored and no zap will occur.

Generally, you can zap by overtyping displayed hex, text, and machine instruction mnemonics.

Example, suppose that the following is displayed at your terminal:

```
_ 003210 -- ARTEST.X2+0
_    .+0 47F0 F01A X2       BC    B'2111',1A(,R15)
_    .+4          15 21                 *.*
_    .+5 E0005MID E7F14040 404040                    *X1     *
_    .+C          F0F361F3 F061F9F4 40               *03/30/94 *
_   .+15          F1F34BF1 F6                        *13.16*
_   .+1A 90EC D00C E0005ZID STM   R14,R12,C(R13)
_   .+1E 181D                   LR    R1,R13
_   .+20 41D0 F034              LA    R13,34(,R15)
_   .+24 50D0 1008              ST    R13,8(,R1)
_   .+28 5010 D004              ST    R1,4(,R13)
_   .+2C 5810 1018              L     R1,18(,R1)
_   .+30 47F0 D048              B     B'1111',48(,R13)
_   .+34 E0005SVA 00000000 00000000 00000000 00000000 *................*
```

In this example, the overwritable fields are shown **hilighted.** Please note the following:

- A **Z** must be typed at the left of every line to be zapped.
- Data lines can be zapped by overtyping either the displayed hex or the displayed text or both. In the event of conflicts, the hex overrides the text on a byte by byte basis.

When Overtyping Text Data:

overtyped text is not upcased.

- The text's leading frame character must be set either to an asterisk (*) or to a vertical bar (|).
- The zapped in text will be translated according to the leading framing character: EBCDIC if it's an asterisk, ASCII if it's a v-bar.
- You can change the encoding by changing the leading framing character - **!!!BUT!!!** see below for an important warning!
- The trailing frame character also must be either an asterisk or a vertical bar, but it also may be nulled out (think "ERASE EOF"), and it need not match the leading frame character.
- You can tell whether Z/XDC is displaying ASCII or EBCDIC by noticing the characters that Z/XDC uses to frame the text in the display. If Z/XDC is using asterisks (*), then it is displaying EBCDIC characters. On the other hand, if Z/XDC is using vertical bars (|), then it is displaying ASCII. EXAMPLES:
  - EBCDIC text is displayed like this: ***...TEXTSTUF..3.h***

  - ASCII text is displayed like this: **|...TEXTSTUF..3.h|**

- If ASCII or EBCDIC text is displayed, and you want to translate that text into the other code, then change the leading frame character from an asterisk to a vertical bar (or from a vertical bar to an asterisk). All text characters on that line will be translated. All nontext codes will be left unchanged. (You need not change the trailing frame character, only the leading one.)

  BUT BE CAREFUL! *ALL* bytes in the text field that display as characters other than periods will be translated. This can lead to some nasty results if you inadvertently translate some data that is not intended to be text. For example, in the **TEXTSTUF** example shown above, the **3** and the **h** would be translated.

  If you do translate text using this method, then in order to properly view the results of the translation, you may have to issue either **DISPLAY +0 EBCDIC** or **DISPLAY +0 ASCII** (or suitable **FORMAT, SHOW** or **WHERE** commands), whichever is appropriate.

When Overtyping Machine Instruction Mnemonics:

- Machine instructions can be zapped by overtyping either the displayed hex or the displayed opcode mnemonics or both. In the event of conflicts, changed mnemonics override the hex.

- If **SET ILC** is in effect, then mnemonic zaps are checked for instruction length: If a new mnemonic names a machine instruction having a different length than the old instruction, then the zap attempt is rejected. For example, an attempt to change a LR instruction to a BALR would be permitted, but changing the LR to an MVC would be rejected. See **HELP COMMANDS SET ILC** for more information.

- **SET ILC** has no effect on zaps to machine instruction opcodes via overtyping the hex data. The check is made only when mnemonic text is overtyped.

- If **SET NOILC** is in effect, then the instruction length check is not performed. Do not, however, think that changing a 2-byte instruction, for example, into a 6-byte instruction will cause Z/XDC to somehow insert 4 bytes of instruction space into your program. That is simply not possible! Instead, the next 4 bytes that follow what had been the 2-byte instruction will now be interpreted as being part of the 6-byte instruction. In most cases, this is not a useful result.

When Overtyping Hex Data:

- Typing a blank, causes the original digit to be restored.

- Ditto for typing an underscore (_).

- Nulling the field causes the zap to be ignored.

- The length of the zap is ended by nulls, but NOT by blanks.

- Trailing nulls/blanks cause the trailing bytes NOT to be zapped or restored.

- Trailing '_'s cause the trailing bytes to be restored.
- Embedded nulls are not possible.
- Typing a digit over an original spacer underscore is not permitted.

Help SHortcutcmds ASTerisk

When the **LIST VARIABLES** command is used to display C variables that contain character strings, c/XDC attempts to show the text as ASCII or EBCDIC according to the current **SET FORMAT ASCII|EBCDIC** setting. For individual variables, This can be overridden as follows:

- The * shortcut forces c/XDC to display the selected variable using the **EBCDIC** codeset.

- The | shortcut forces c/XDC to display the selected variable using the **ASCII** codeset.

Help SHortcutcmds |

When the **LIST VARIABLES** command is used to display C variables that contain character strings, c/XDC attempts to show the text as ASCII or EBCDIC according to the current **SET FORMAT ASCII|EBCDIC** setting. For individual variables, This can be overridden as follows:

- The | shortcut forces c/XDC to display the selected variable using the **ASCII** codeset.

- The * shortcut forces c/XDC to display the selected variable using the **EBCDIC** codeset.

Help SHortcutcmds \

Help SHortcutcmds /

When used on a **LIST VARIABLES** display line that is showing a character array:

- The / shortcut causes c/XDC to display the variable using an **array** paradigm.

- While the \ shortcut causes c/XDC to display the variable using a **string** paradigm.

For a detailed discussion of string vs array formats, as well as examples, see **HELP SHORTCUTCMDS \.**

Help Pascmds

Point-and-Shoot commands (referred to herein as **PaS** commands) are short 1- and 2-character command strings that can be overtyped onto either data fields (including register displays) or machine instructions. They are used to display storage being either pointed to or referenced by the field, register or instruction.

With respect to machine instructions, PaS commands are supported on **all** instructions that in one way or another reference storage, **regardless** of the way in which said storage is referenced!

Briefly...

PaS commands, when typed into a data field or a machine instruction operand, cause an action to occur at the **target** of that data field or instruction operand.

The following PaS commands cause the following Z/XDC commands to be issued at the target location:

D - **DISPLAY**
F - **FORMAT**
I - **LIST LOCATION**
L - **LIST FIXED**
S - **SHOW**
T - **TRAP**
V - **LIST VARS**

The above list will get you started, but there is more to it than this. For example, there is an optional second character that can be used to override the **width** (i.e. the AMODE) of the pointer to be followed. For complete details, see **HELP PASCMDS SYNTAX.**

More Information

The following detailed information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

SYNTAX - The syntax of the many PaS commands
DATAFIELDS - Using PaS commands on registers and data fields

INSTRUCTIONS - Using PaS commands on machine instructions

Help Pascmds Syntax

Point-and-Shoot commands (referred to herein as **PaS** commands) are short 1- and 2-character command strings that can be overtyped onto either data fields (including register displays) or machine instructions. They are used to display storage being either pointed to or referenced by the field, register or instruction.

Single Character PaS commands:

The following single character PaS commands define an action that is to occur at a target location, but the resolved target address will be trimmed to a default width (AMODE):

- Most of the commands cause a storage displaying action of some sort.
- One of the actions sets traps.
- In all cases, the pointer to the target location is considered to be either 24, 31, or 64 bits wide according to the user program's current addressing mode as displayed by the **LIST AMODE** command. (You can override this by using a two-character PaS command. See below.)

D
The location pointed to is displayed in a raw hex-text (dump like) format via Z/XDC's **DISPLAY** command.

F
The location pointed to is displayed with formatting via Z/XDC's **FORMAT** command.

I
The location pointed to is processed to determine if Z/XDC can identify the location.

L
The location pointed to is displayed with formatting via Z/XDC's **LIST FIXED** command.

S
The location pointed to is displayed with formatting via Z/XDC's **SHOW** command.

T
A transient breakpoint (like what a **TRAP** command sets) is set at the location **pointed to** by the field into which the **T** is placed.

V
The location pointed to is examined to determine if it contains a High Level Language variable. If it does, the contents of that variable is displayed via Z/XDC's **LIST VARIABLES** command.

Special Note: When the **D** or **F** PaS command is used by itself, and it is typed directly onto a matching D or F hex digit, then you must follow the **D** or **F** command

with a blank; otherwise Z/XDC won't realize that you're trying to enter a PaS command.

The following single character PaS commands override the width of the resolved target address, but leave the action to be taken to a default:

? or **%** or **!**
The location pointed to is displayed via Z/XDC's **FORMAT** command or **DISPLAY** command according to the command that was used to display the location into which the pointer itself was entered.

The resolved pointer is trimmed to a width of either 24, 31, or 64 bits as follows:

? - The pointer is considered to be 31 bits wide.
% - The pointer is considered to be 24 bits wide.

! - The pointer is considered to be either 24, 31, or 64 bits wide according to the current **SET BANG** setting (as displayed by the **LIST BANG** command). See **HELP COMMANDS SET BANG** for more information.

Two Character PaS commands:

The **D F L S T V** and **W** PaS commands can be used in combination with the **? %** and **!** PaS commands to fully control both the target resolution width (AMODE) and the action taken. Note, the two characters can be given in either order.

Example: The PaS command **F% (or %F)** truncates the resolved target address to 24 bits and displays the target location using Z/XDC's **FORMAT** command.

Resolving the Target Location

The nature of Point-and-Shoot commands is that they perform their actions, not at the location where they are set, but at the location **pointed to** by the field in which they are set.

PaS commands can be placed into either of two types of pointer fields:

Machine Instructions

- If the receiving field is part of a displayed **machine instruction,** then storage is displayed (or a trap is set at) the location that is **pointed to** by that instruction. In this regard, Z/XDC's understanding of all the ways machine instructions reference storage is **comprehensive!**

- It matters where, within the machine instruction, the PaS command is placed. So if an instruction references two storage locations, Z/XDC will know which reference you intend the PaS command to follow.

- Altogether, there are four "standard" ways (RS, RX, RI and SS) and at least **55 non-standard** ways that instructions make their storage references. **Z/XDC understands them all!** See **HELP PASCMDS INSTRUCTIONS** for more information.

Data Fields

- If the receiving field is a hex data field, then the contents of that field are treated as a pointer. The width of the pointer is determined as described above. Storage is displayed (or a trap is set at) the location that is **pointed to** by that data field. See **HELP PASCMDS DATAFIELDS** for more information.

Using Multiple PaS Commands

Some PaS commands are various forms of storage displaying commands (D F and V). While you **can** set multiple of these prior to pressing ENTER, it is generally pointless to do so. Yes, a display for each command placed will be generated, but only the **first** of these will be visible. You would have to **scroll down** to see the rest.

Some PaS commands have a small set of output lines (L S and I). You **can** set multiple of these prior to pressing ENTER. Depending on the window size, you may be able to see the display from all the generated commands. If the window is not large enough, You would have to **scroll down** to see the rest.

The **T** PaS command is a different story. Generally, the best use case for **T** involves placing them on several machine instructions before pressing ENTER:

- For each **T** placed, a transient breakpoint will be created at the instruction's target address.
- All breakpoints set by a single ENTER will be assigned to the same **family.** This means that when any one of them is reached by execution, they **all** will be automatically removed.

A comprehensive example of all this can be found at **HELP PASCMDS INSTRUCTIONS T-EXAMPLE.**

More Information

The following additional information is available. Type an **H** at the left to select directly.

- **HELP PASCMDS DATAFIELDS:** Using PaS commands on registers and data fields
- **HELP PASCMDS INSTRUCTIONS:** Using PaS commands on machine instructions

Help Pascmds Datafields

Point-and-Shoot commands (referred to herein as **PaS** commands) are short 1- and 2-character command strings that can be overtyped onto either data fields (including register displays) or machine instructions. They are used to display storage being

either pointed to or referenced by the field, register or instruction. The syntax of PaS commands can be found in **HELP PASCMDS SYNTAX.**

Using Point-and-Shoot (PaS) Commands on Data Fields

When a window is displaying registers or storage, the display frequently includes 1-, 2-, 3-, 4- and 8-byte long groups of hexadecimal values. If such a group contains an address of a storage location, then it is a "pointer field". If the location being pointed to is of further interest to you, then you can display that location by tabbing the cursor over to the pointer field and then overtyping the first one or two digits with one of the PaS commands described in **HELP PASCMDS SYNTAX.**

8-Byte Wide Pointers

8-byte wide pointers are supported even when such pointers are displayed as two adjacent 4-byte fields. If you use the **!** PaS command, Z/XDC will know to use 8 bytes of storage starting with the byte onto which you typed the **!.**

Position Sensitivity - Data Fields

When used in hex **data** fields (machine instruction hex fields are processed differently), Point-and-Shoot commands are always position sensitive:

- All hex digits within the data field that are to the **left** of the PaS command are **ignored!**
- The PaS's target address is built from the digits starting at the location of the start of the PaS command, and running rightwards for as many digits as necessary.
- As many digits are used as necessary **regardless** of whether or not the needed digits:
  - Are displayed within the current field,
  - Are displayed within the next field on the same line,
  - Are displayed in the first field on the following line,
  - Or are not displayed at all!
- The number of digits required for a PaS command is as follows:
  - For **64-bit** indirect (!), **16 digits** are used to form the target address.
  - For **31-bit** indirect (?), **8 digits** are extracted, and then the lo-order **31 bits** are used to form the target address. (I.e. the hi-order bit is ignored.)
  - For **24-bit** indirect (%), **8 digits** are extracted, and then the lo-order **3 bytes** are used to form the target address. (I.e. the first two extracted digits are ignored.)
  - **Exception:** For **24-bit** PaS commands (%):

- If the field containing the PaS command is exactly **6 digits** (3-bytes) wide,
- And if the PaS command starts on the **leftmost** digit,
- And if the 6-digit field is the **only** field displayed by the current display line,

Then those 6 digits are used to form the target address. (i.e. in this exception, the leading 2 digits are **not** ignored.)

Example - Using PaS Commands on Hex Data Displays

Note, the following examples are pretty complicated, so it might be easier if you just tried the various PaS commands yourself in your own displays. Nevertheless, here goes...

Suppose that the following display lines are currently in view on your screen:

```
D R13?+9 B=2F
00436FC1  8p   425B20 01527C98 0042EB1C 00000018   *.$...@q........*
00436FD0  8p 0032692C 00E82000 0042FA78 0157FB24 *.....Y.........*
00436FE0  8p 7D1A7FA0 9527B3AC 0032260B 00000000  *...............*
D R13?+9+4 B=3
00436FC5  8p            527C98 *.@q*
```

Suppose also that **SET BANG 64BIT** is in effect (i.e. the **!** indirect operator is currently set to represent 64-bit addressing [not AMODE sensitive addressing]).

And suppose that the retry level PSW's current AMODE is 31.

Then the following shows what happens when various PaS commands are overtyped at various positions with the fullword (above) that contains **0157FB24.**

P-a-S Cmd	First Overtyped Digit in 0157FB24	Display Address	Display Type	Comment
F 1st (0)	0157FB24	FORMAT	Digits are drawn from subsequent fullwords as needed.	
F 2nd (1)	157FB247	FORMAT	Digits to the left are ignored. Digits needed on the right are brought in from the following fullword (from 7D1A7FA0, in this case)	
F 3rd (5)	57FB247D	FORMAT	Ditto	
F 4th (7)	7FB247D1	FORMAT	Ditto	
D 5th (F)	7B247D1A	DISPLAY	Ditto	
F 5th (F)	**Nothing happens!** Because Z/XDC can't see F on F.			
F-blank 5th (F)	7B247D1A	FORMAT	The trailing blank causes Z/XDC to realize that the preceding **F** is a PaS command, not just a hex digit.	
S% 1st (0)	0057FB24	SHOW	% trims the target address to 24 bits.	
S% 2nd (1)	007FB247	SHOW	% Extracts 8 nibbles (4 bytes) starting with the **nibble** on which the PaS command starts. (The trailing **7** is taken from the next following fullword.)	
? 4th (7)	7FB247D1 **DISPLAY** Because it's from a **D...** display.			

```
! 1st (0)     0157FB24_7D1A7FA0   16 digits are needed for 64-bit, -
! 2nd (1)     157FB24_7D1A7FA0_9   so additional digits are taken -
! 5th (F)     FB24_7D1A7FA0_9527    as needed from the next -
! 8th (4)     4_7D1A7FA0_9527B3A     following field.
% on 425B20 005B2001            Because the display line includes multiple
                                        fields. [The trailing 01 is pulled in from
                                        the next field.]
% on 527C98 00527C98            Because the display line shows only a
                                        single, 3-byte field.
```

More Information

The following additional information is available. Type an **H** at the left to select directly.

- **HELP PASCMDS SYNTAX:** The syntax of the many PaS commands
- **HELP PASCMDS INSTRUCTIONS:** Using PaS commands on machine instructions

Help Pascmds Instructions

Point-and-Shoot commands (referred to herein as **PaS** commands) are short 1- and 2-character command strings that can be overtyped onto either data fields (including register displays) or machine instructions. They are used to display storage being either pointed to or referenced by the field, register or instruction.

With respect to machine instructions, PaS commands are supported on **all** instructions that in one way or another reference storage, **regardless** of the way in which said storage is referenced!

Using PaS Commands on Machine Instructions

When formatted machine instructions are displayed, the hex equivalents of the displayed instructions are shown as one or more halfwords to the left. Depending upon the particulars of the specific machine instruction being formatted, the displayed halfwords may contain s-cons, relative immediate fields, one or more specified or implied registers that point to storage locations, or whatever else IBM has dreamed up. Regardless of the storage reference method used, **Z/XDC understands them all!**

So when it makes sense to do so, PaS commands can be used to display storage (if any) pointed to by **any** machine instruction. Just tab the cursor over to the halfword of interest, and then overtype the appropriate one or two digits with one of the PaS commands described in **HELP PASCMDS SYNTAX.**

Note, those hwords that accept PaS commands are displayed **hilighted.** Those that do not are dim-lighted.

When you use a PaS command, Z/XDC identifies the relevant registers (and storage

reference method) based on the position of your PaS command within the machine instruction. Then it resolves the command using the **current values** found within the **retry level** instances of those registers.

WARNING - The Register Contents Need to be Right!

When resolving Point-and-Shoot commands, Z/XDC will almost always have to use the contents of registers (and sometimes the PSW) in order to calculate the storage locations being referenced. So Z/XDC obtains the necessary data from the values **currently** contained in the **retry level** registers and PSW.

Obviously, **if the registers change** between "now" and the point in time when execution reaches the instruction being examined, the resolutions being done "now" **will be wrong!** So you do have to be careful when using PaS commands. Specifically:

- If the instruction on which you place a PaS command is **exactly at** the current retry level execution point, then the PaS resolution will be correct (guaranteed).

- If the instruction on which you place a PaS command is **near** the current retry level execution point, then the PaS resolution **may** be correct or it **may be wrong,** depending upon whether or not the "now" values of the necessary registers will still be the same when execution eventually reaches that instruction.

- If the instruction on which you place a PaS command is in an **entirely unrelated** section of code (such as in a different csect or load module), then the PaS resolution is almost certainly **guaranteed to be wrong!**

- For **PC** instructions, especially those generated by IBM macros, it is almost always the case that the necessary PC# is loaded into a register by the next prior instruction to the PC instruction. Therefore, a PaS command on a PC instruction will almost always produce an **incorrect result** unless the retry level execution point is exactly at the PC instruction.

Examples of Instructions on Which PaS Commands Can be Used

Z/XDC’s understanding of machine instructions is comprehensive! Pretty much any way in which a machine instruction might use operands as pointers to storage is understood by Z/XDC. Here are some expected and (perhaps) unexpected examples:

- **L STH, J, LA, LARL, BE, JCT, BXLE, LY, MVC** (both operands) **and so on and so on and so on...** Z/XDC understands **all** the ways instructions reference storage including all the normal ways used by RS, RX, RI and SS instructions.

- **CFC** (compare and form codeword) uses implied general registers 1 and 3 as storage pointers. Z/XDC knows that.

- **BAKR** (branch and stack) Both the R-one and R-two operands point to storage. So

for instructions such as this, **where** the PaS command is placed (within the displayed machine instruction's 2nd halfword) matters. (Ditto for **MVCL** etc.).

Further, in the case of BAKR, with respect to the R-one operand, the width of the pointer (24 bits, 31 bits or 64 bits) depends, not on the current AMODE, but on bits in the R-one register itself. Z/XDC knows this!

- **ESTA** (extract stacked state) extracts data from the current linkage stack entry. Z/XDC knows how to find and display that entry.

- **RRBE** (reset reference bit extended) resets the reference bit for blocks of real storage. Z/XDC knows how to find and display those blocks **in real storage!**

- **PC** (program call) passes control to a service routine located, potentially, in another address space. Z/XDC knows how to find the ASTEs and ETEs necessary to determine the location (address and aspace) of the service routine being called. (But don't forget the caveat about PCs mentioned amongst the WARNINGs above.)

- **SVC** (supervisor call) resolutions will almost always be right because the resolution is governed by a table in the System Nucleus which Z/XDC knows how to find and use.

  The only time the target of an SVC depends upon a register is when the SVC is EX'd, in which case, a PaS resolution will never be right.

  What will be right, though, is the resolution that occurs when execution has **reached** an EX (or EXRL) of an SVC instruction. At that point, Z/XDC will know exactly what the SVC is and where its handler resides in storage. [See **HELP BREAKPOINTS TRACING EX** for more information.]

WARNING! Z/XDC's resolution of PaS commands generally will be useful **only** if the contents of the retry level general registers currently are the **same** as they would be when program execution reaches the machine instruction being examined. For example, operands in machine instructions located in load modules or programs having nothing to do with the currently executing user program **will most certainly not be resolved correctly** by Z/XDC.

Cursor Sensitivity - Machine Instructions

When used in a machine instruction's hex displayed halfword fields, PaS commands need to be placed within the correct halfword field, but precisely where within such fields generally does not matter **unless it has to matter:**

- If individual digits within a halfword reference differing areas of storage, then cursor position matters.

- On the other hand, if all digits within a halfword are part of the same storage referencing operand (or are uninvolved in storage references), then cursor position doesn't matter.

For example, if a PaS command is placed upon the hex interpretation of an **MVCL** instruction, then the cursor position of the command has the following effects:

- If placed on either of the instruction's first two digits, the command is **rejected** as being incorrectly placed.

- If placed on the **3rd** digit, then the location pointed to by the instruction's **first** register (the sink location) is displayed.

- If placed on the **4th** digit, then the location pointed to by the instruction's **second** register (the source location) is displayed.

More Information

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

↕ **F-EXAMPLE** - Examples of using the **F** (and D) PaS command on machine instructions.
↕ **T-EXAMPLE** - Example of using **T** PaS commands to allow a loop to run to completion without losing control.

The following additional information is also available. Type an **H** at the left to select directly.

↕ - **HELP PASCMDS SYNTAX:** The syntax of the many PaS commands.
↕ - **HELP PASCMDS DATAFIELDS:** Using PaS commands on registers and data fields.

Help Pascmds Instructions F-example

Point-and-Shoot commands (PaS commands) provide an easy way to display storage that is pointed to by machine instructions. When a PaS command is placed in a machine instruction's displayed hex code, Z/XDC decodes the instruction's storage reference and displays the storage pointed to by that instruction.

The z/System instruction set currently has 59 distinct ways to reference storage. Z/XDC understands all of them.

↕ For full details, and an **important warning** regarding incorrect register contents, see **HELP PASCMDS INSTRUCTIONS.**

Examples of Using PaS Commands on Machine Instructions

Let's examine an example. Suppose that the following display lines are currently in view at your terminal.

```
F .+58
3A4968 --- XDC.DBCHELP+58
3A4968  184F                      LR    R4,R15
3A496A  4730 807A BC    B'0011',7A(,R8)
3A496E 0743 BLR   R3
3A4970  4100 F000 LA    R0,0(,R15)
3A4974  4810 806C LH    R1,6C(,R8)
3A4978  47F4 8070 BC    B'1111',6E(R4,R8)
3A497C           C504                                  *E.*
3A497E                    CONTU    EQU   *
```

- Tabbing over to the **807A** halfword (part of the first BC instruction) and overtyping the 80 with a **D** or an **F** causes a new display to be generated showing the storage located at **R8~INDIRECT(AMODE)+7A.** The new display will be formatted by Z/XDC's **DISPLAY** or **FORMAT** command according to whether a **D** or an **F** was typed. [Follow the link for information about the **INDIRECT()** built-in function.]

- Alternatively, overtyping that field with a **?** **%** or **!** causes location **R8?+7A, R8%+7A,** or **R8!+7A** to be displayed via Z/XDC's **FORMAT** command (because the original display, shown above, was itself generated by a FORMAT command).

- Tabbing over to the **184F** (the LR instruction) and overtyping it with anything is **invalid** because the context indicates to Z/XDC that the **184F** neither is an s-con nor contains any registers being used as pointers.

- On the other hand, tabbing over to the **0743** halfword (the **BLR** instruction) and overtyping it anywhere with a PaS command **is valid** since in this context, Z/XDC recognizes that **R3** is a pointer. (It points to a potential branch target.) So the location pointed to by **R3** will be displayed.

- Tabbing over to the **8070** (the second BC instruction) and overtyping it with a PaS command causes location **R8?+X4(R4)+70** to be displayed. In other words, the PaS command will take into account index registers when appropriate. [Follow the link for info about the **X4()** built-in function.]

- Tabbing over to the **C504** (at location 3A497C) and overtyping it with a PaS command will be processed by Z/XDC as a data reference because the context indicates to Z/XDC that the **C504** is not an instruction. The result will be a display of location **0000C504.**

Help Pascmds Instructions T-example

The **T** Point-and-Shoot command (PaS command) provides an easy way to set a transient breakpoint onto the target of a branching instruction. When a **T** is placed in a branch's (or jump's or whatever's) displayed hex code, Z/XDC decodes the instruction's storage reference and sets a trap at the target location that the branch might jump to.

The z/System instruction set currently has 59 distinct ways to reference storage. Z/XDC understands all of them. For full details, and an **important warning** regarding incorrect register contents, see **HELP PASCMDS INSTRUCTIONS.**

Note, Z/XDC regards **any** instruction that can change the flow of execution to be a branching instruction. This includes branches, jumps, SVCs, PCs, LPSWs, LASPs, BAKRs, and so on.

What follows is an example of a very good use case for **T** PaS commands.

Letting a Loop Run Without Losing Control (a **T** PaS Use Case)

When stepping through code with the **TRACE** command, you will inevitably encounter loops that will iterate dozens, hundreds or even millions of times! So naturally, you will want to let these loops just run to completion and recapture execution when they pop out at the end.

Simple Loops

For example, suppose you have a nice simple **JCT** loop, and you've already stepped down the loop's first pass and now execution is poised at the bottom on the **JCT** itself. What's next?

Well, if you simply issue **T** (F9) or **T BY** (F10), you will continue into the 2nd pass.

But on the other hand, if you want to just let the loop run, and catch it when it falls through the JCT, you need to:

- Set a **TRAP** on the instruction following the JCT.

- Then issue a **GOT** to let the loop run.

Actually, **F11** will do exactly that. It will set temporary traps on the next several following instructions, and then it will issue a **GOT.** Then when execution falls through the **JCT** and reaches the first of the traps, all of the traps will be removed.

So, while **F11's** normal use case is for stepping over subroutine calls, it also comes in quite handy at the bottom of simple loops.

But for More Complicated Loops...

While **F11** can sometimes be quite handy, for most loops it's not enough...

This setting a fall-thru trap and issuing **GOT** works very nicely for simple loops that always end by falling through the bottom. But what about a more complex case where the loop does not always run to completion, but instead jumps out to error handlers, special conditions processing or who knows what all else? In these cases, you will need to put traps at the targets of all branching instructions contained within the loop. That's where the **T** PaS instruction comes in real handy!

- First, use a **FORMAT** command (or **WHERE)** to display all (or even part) of your

loop.

- Identify all branching and jumping and what-not'ing instructions that can take execution out of the loop.

- Use the **T** PaS command on each of those instructions to set traps at each of their branch/jump **targets.**

- Also, use the T **shortcut** command to set a trap past the bottom of the loop.

- Be sure to **avoid** using the T PaS commands on branches that **stay within** the loop!

If you've done it right, then when you issue **GOT,** the loop will run until it either completes or exits, and the next thing you will see is a display of your code at the first instruction reached outside of the loop.

A Loop Control Example

The following code snippet shows a very small and fairly simple loop.

```
                  ****************************************
                  * This is TSO. Issue a TPUT. First, if   *
                  * requested, then strip off the message  *
                  * id.                                    *
                  ****************************************
_ .+550 9120 B278 TM   DSATFLAG,NOMSGID Suppress message id?
_ .+554 4780 C568 BZ   PUTM1MID         No, skip

                  ******** Top of Loop *****************
_ .+558 91BF 1000 PUTM1MIL TM   0(R1),255-C' '   Yes, blank reached yet?
_ .+55C 4780 C568 BZ   PUTM1MID         Yes, msgid stripped
_ .+560 4110 1001 LA   R1,1(,R1)        No, advance scanner
_ .+564 46F0 C54C BCT  R15,PUTM1MIL     Loop for next character
                  ******** Bottom of Loop **************

_ .+568 5810 B054 L    R1,SAVEPUM1+12   No blanks; Restore msg ptr
_ .+56C 43F0 1000 IC   R15,0(,R1)       Restore L'msg
_ .+570 4110 1001 LA   R1,1(,R1)        --> text
_ .+574 91BF 1000 PUTM1MID TM   0(R1),255-C' '   Blank here?
_ .+578 4770 C57C BNZ  PUTM1PUT         No, go issue TPUT
```

There are two ways this loop can end:

- It can fall through the **BCT** at **.+564**
- It can take the **BZ** at **.+55C**

So if I want to let this loop run and recapture control when it completes, I'll need to set two traps:

- One at **.+568,** following the **BCT.**
- One at **.+574,** the target of the **BZ.**

The easiest way to do this is as follows:

- Place a **T** on the BZ's **C568** field.
- Place a **T** on the shortcut field **[_]** at far left of the **L** instruction following the BCT.

```
    _ .+55C 4780 T568 BZ   PUTM1MID       Yes, msgid stripped
                               [...]
    _ .+564 46F0 C54C          BCT  R15,PUTM1MIL   Loop for next charac
    T .+568 5810 B054          L    R1,SAVEPUM1+12 --> text
```

- Press **ENTER.**

The following happens:

- Two temporary traps will appear:
  - One at **.+574,** the target of the **BZ.**
  - One at **.+568,** where you set the shortcut **T.**
- Both traps have the same family ID (the numeric part of the trap names). This means that when either is reached by execution, they both will be cleared.

```
    _ .+558 91BF 1000 PUTM1MIL TM   0(R1),255-C' '   Yes, blank reached
    _ .+55C 4780 C568          BZ   PUTM1MID         Yes, msgid strippe
    _ .+560 4110 1001          LA   R1,1(,R1)        No,advance scanner
    _ .+564 46F0 C54C          BCT  R15,PUTM1MIL     Loop for next char
    _ .+568 0010 B054 TR00008C L    R1,SAVEPUM1+12   No blanks; Restore
    _ .+56C 43F0 1000          IC   R15,0(,R1)       Restore L'msg
    _ .+570 4110 1001          LA   R1,1(,R1)        -->text
    _ .+574           TR00008D EQU  *
    _ .+574 00BF 1000 PUTM1MID TM   0(R1),255-C' '   Blank here?
```

Now when you issue **GOT:**

- The loop will run to completion.
- Z/XDC will recapture execution when it reaches (in this case) the **TR00008D** trap at **.+574.**
- Both traps will be removed.
- The code will be displayed, showing the resume execution location now to be at **.+574.**

```
    _ .+558 91BF 1000  PUTM1MIL TM   0(R1),255-C' '   Yes, blank reache
    _ .+55C 4780 C568  @BEA     BZ   PUTM1MID         Yes, msgid stripp
    _ .+560 4110 1001           LA   R1,1(,R1)        No, advance scann
    _ .+564 46F0 C54C           BCT  R15,PUTM1MIL     Loop for next cha
    _ .+568 5810 B054           L    R1,SAVEPUM1+12   No blanks; Restor
    _ .+56C 43F0 1000           IC   R15,0(,R1)       Restore L'msg
    _ .+570 4110 1001           LA   R1,1(,R1)        --> text
    _ .+574           >CDP      EQU  *
```

```
_ .+574 91BF 1000 >PUTM1MID TM   0(R1),255-C' '   blank here?
```

Help DEbugging

Z/XDC is a versatile tool that can be used to debug a wide variety of programs running in a wide range of environments. This is because Z/XDC is designed to run as an "ABEND error recovery" routine, i.e. an ESTAE, ESTAI, ARR or FRR routine. Consequently, Z/XDC can be used to debug nearly any program running in environments where ABEND recovery routines can be used. The only restriction is that at the time that an ABEND occurs or a breakpoint is reached by your program, Z/XDC must (with rare exceptions) be the newest recovery routine in the environment and, therefore, first in line to receive control to handle the ABEND (or breakpoint).

Z/XDC can also run as a **Trap Handler.** However even so, that does not change the fact that Z/XDC still must also be set up as your execution environment's newest ABEND recovery routine. There are many reasons for this. See HELP BREAKPOINTS TYPES for the details.

Making sure that Z/XDC is the newest recovery routine can be simple or complex depending upon the structure of the program that you are trying to debug. Sometimes you will have to modify your program in order to achieve this. Most times, you won't. If you do have to modify your program, then sometimes you may need to modify it only at execution time when you first start up your debugging session. Other times, you may find it more convenient to modify your program at assembly time so as to add a permanent Z/XDC interface to your code.

Subtopics Index

The following topics explain the specific considerations for setting up Z/XDC interfaces for debugging various kinds of programs running in various environments. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

GETTINGSTARTED - Setting up a program for debugging by Z/XDC.
AMODE64 - Debugging 64-bit programs.
BATCHJOBS - Debugging programs running as batch jobs or system tasks.
C - Debugging programs written in XL C/C++ and Metal C.
CICS - Debugging transactions and exit routines running within CICS.
DUMPS (new) - Using Z/XDC to examine IPCS dumps
ESTAES - Using Z/XDC in programs that already have their own ABEND error recovery routines.
EXITROUTINES - Debugging system and product exit routines. Also debugging programming exit routines such as attention exits, IRB exits, DCB exits, timer exits, etc.
FRR - Running Z/XDC as an Functional Recovery Routine (FRR) to debug task mode and SRB mode code.
HOOKS - This topic has been moved to **HELP HOOKS.**
ISPF - Using Z/XDC in ISPF environments. Also, debugging ISPF dialogs.
JES2 - Debugging JES2 exit routines and main task code.

LE	- Debugging Assembler programs running within IBM's Language Environment.
LOSTLOCKS	- Debugging programs that hold locks.
PCROUTINES	- Debugging PC routines.
PLPA	- Debugging programs located in the PLPA and other common storage areas.
REENTRANT	- Debugging reentrant (RENT) load modules and program objects.
REFRESHABLE	- Debugging refreshable (REFR) load modules and program objects. (REFRPROT issues)
RMODE64	- Debugging programs located Above the Bar.
SRBMODE	- Debugging SRB mode routines.
TSO	- Debugging TSO commands and other programs running in TSO.
USS	- Debugging programs in a USS (Unix System Services) environment.
TIPS	- Miscellaneous tricks, hints and tips.

Help DEbugging Gettingstarted

Z/XDC is a very powerful tool for debugging a wide variety of programs ranging from the simple to the complex. This is because Z/XDC is designed to run as an "ABEND error recovery" routine, i.e. an ESTAE, ESTAI, ARR or FRR type routine. Consequently, Z/XDC can be used to debug nearly any program running in environments where recovery routines can be used. The major restriction is that at the time that an ABEND occurs or a breakpoint is reached by your program, Z/XDC must be the newest recovery routine in the environment and, therefore, first in line to process ABENDs or breakpoints.

c/XDC is an Optional Feature of Z/XDC, so any reference below (and elsewhere) to the **/Z/XDC product** should be read as including c/XDC.

Major Parts of Z/XDC

The Z/XDC product has many components, but for the purposes of this introduction, the two major pieces are the load modules named **XDC** and **XDCCALL.**

- The load module named **XDC** is Z/XDC's primary load module. It is Z/XDC's main program. It is a program that is designed to run as an ABEND error recovery routine. It receives control from z/OS's "Recovery/Termination Manager" (RTM) when ABENDs occur or breakpoints are executed, and it displays information to your terminal and accepts and executes your commands. In fact, the most basic way for a program to setup Z/XDC is for it to issue a **LOAD** macro to bring Z/XDC into storage and an **ESTAE(X)** macro (or **SETFRR** macro) to set Z/XDC up as your program's ABEND error recovery routine. Example:

```
          LOAD  EPLOC==CL8'XDC',ERRET=NOXDC   ("==" is not a typo)
          ESTAE (R0)
    NOXDC DS    0H
```

  (If Z/XDC is installed under a clone name [Example: V31], then change =CL8'XDC' to match the clone name ["=CL8'V31'"]. For information about renaming Z/XDC, see HELP XDCCLONES.)

(Note, throughout Z/XDC documentation, clone names are represented generically as **xxx.)**

(Also, see the discussion of the **//XDCISxxx DD DUMMY** allocation in HELP DDNAMES XDCIS for ideas about making an XDC interface that is independent of XDC's actual name.)

- The **XDCCALL** load module provides a higher level interface to Z/XDC. It creates a debugging environment within which a user program can be executed. XDCCALL accepts information about a program's name, its parameters if any, its load library(s), and its authorization level, and then it attaches that program as a subtask and establishes Z/XDC as an ESTAI type recovery routine for the subtask. This means that many programs (but not all programs) can be debugged by Z/XDC **without** requiring any further setup on your part.

But when further setup is required, using **Hooks** makes the process very simple.

Note, Z/XDC has a Startup Panel in ISPF (named **XDCPANEL)** that starts TSO resident debugging sessions by invoking XDCCALL.

XDCCALL can also, of course, be invoked in the batch via JCL.

Getting Z/XDC Into Your Program

As indicated above, sometimes, in order to use Z/XDC, it is necessary that the user program be modified to establish Z/XDC as its ESTAE routine. Sometimes such modifications can be done at execution time at the start of the debugging session. Other times, it is more appropriate to build a permanent Z/XDC interface with source code changes.

In other cases **(and most easily),** establishing a Z/XDC debugging session can be done externally by Z/XDC's **HOOK** command, thus avoiding the need to modify the user program in any way.

Whether or not a program needs to be modified depends upon the complexity of the program with regards to its usage of z/OS environments and interfaces. If it is necessary to modify a program in order to use Z/XDC, it is **not** always necessary that the modifications be made by changing the program's source code. Often it is possible to make the modifications at program execution time. There are at least two reasonably good approaches to take:

- Z/XDC has a very powerful storage ZAP command that can be used to make appropriate changes to your program at the very beginning of a debugging session. Example: If your program issues an ESTAE macro, then you can use Z/XDC's ZAP command to change the generated SVC 60 instruction (X'0A3C') to a less troublesome "SR R15,R15" (X'1BFF').

(Z/XDC has the ability to read and execute command scripts from a dataset, so even complex "startup zaps" can be feasible to use.)

- **A better alternative** would be to use Z/XDC's **HOOK** command to create a suitable Z/XDC debugging environment at any point in your code that you want to debug. In essence, the HOOK command inserts into your code a bit of logic that jumps execution over to Z/XDC's Hook Support code, which establishes Z/XDC as the

newest recovery routine (ESTAE-type or FRR) for the current execution environment. Then a breakpoint is executed to pass control to a Z/XDC debugging session.

If the program that you want to debug is structurally simple (in terms of its use z/OS interfaces and environments), then that program can be debugged by Z/XDC without modification. Just run the program under the control of XDCCALL.

Using Z/XDC in Difficult Environments

If on the other hand the program has any of the following characteristics, then modifications may be necessary in order to use Z/XDC to its fullest capabilities:

- Some programs run in environments such that it is not possible to start them with XDCCALL. Examples:
  - SMF exits
  - DASDM exits
  - TSO logon exits
  - JES2 exits
  - Other z/OS system exits
  - CA-ACF2 exits
  - Exits of other products that run as z/OS system components

- Other programs run as exits within products or programs that establish their own ESTAE/ESTAI routines. So while it is possible to run the product under XDCCALL, by the time your code is called, the product has already created its own ESTAE/ESTAI's routines that are newer than the ESTAI that XDCCALL created. Examples:
  - TSO SUBMIT command exits
  - IOF exits
  - SDSF exits

  When this is the case, you will want to use the **HOOK** command to insert a debugging session into the code.

- If a program already has recovery routines of its own, then it will have to be modified in order to use Z/XDC.
  - Either the ESTAE/ESTAIs will have to be removed. (The ZAP command can do this.)
  - Or they will have to be superseded. (The HOOK command can do this. This is the easiest way.)
  - Or the ESTAE/I exit routines themselves will have to be modified to call Z/XDC under appropriate conditions. (For particularly complex products, this may actually be the best longterm solution. See HELP DEBUGGING ESTAES MODIFYING for more information.)

- If a program has components that run under separate z/OS Request Blocks (RBs), then those components will have to be individually HOOK'd if Z/XDC is to be used to its fullest capabilities for debugging them. Examples:
  - DCB exits (OPEN, LABEL, EOV, etc.)
  - STIMER exits
  - Attention exits
  - VTAM IRB exits
  - ETXR exits ("End of Task eXit Routines")
  - Subsystem exits

If these routines are **not** HOOK'd, then Z/XDC may still be used to debug them but only in a limited way: Z/XDC will intercept ABENDs and breakpoints that occur within such components, but Z/XDC will not be able to resume their execution. Consequently, you will be able to examine ABENDs, but you will not be able to use the TRACE command, for example, to step through them instruction by instruction.

Debugging Recovery Routines, SRB Routines and Multi-Tasking Routines

Z/XDC can also be used to debug the user program's own ESTAE(X)/ESTAI routines, and (perhaps unexpectedly) modifications to the user program are **not** necessary in order to do this. Just start your program under XDCCALL, set your breakpoints into your program's ESTAE/ESTAI routine, create the necessary conditions for your program to ABEND, then GO. When your program ABENDs, its ESTAE/ESTAI routine will gain control. Then when the ESTAE/ESTAI routine reaches the breakpoint, Z/XDC will gain control. This is because in z/OS, ESTAIs (which is what Z/XDC is when XDCCALL is used) can request program retry either for "main programs" or for ESTAE routines. In essence, ESTAIs automatically function as ABEND error recovery routines for ESTAEs. For more detailed information about this, see HELP EXECUTIONLEVELS RETRYLVL ESTAIS.

Multitasking programs also can be debugged without modification so long as none of the above discussed conditions occur. This is because z/OS automatically propagate ESTAIs to subtasks; therefore, the Z/XDC ESTAI that XDCCALL creates is automatically propagated to all descendant tasks of the program being debugged.

Z/XDC can be used to debug not only task mode programs, but **SRB mode** routines as well. See HELP DEBUGGING SRBMODE for details.

Understanding Recovery Environments and Request Blocks

IMPORTANT: "Request Blocks" ("RBs") are z/OS control blocks that identify and control the execution of programs that are running within a task. They serve as save areas for a program's PSW and registers and for such status information as whether or not a program is executing in supervisor state. Also, RBs have an "ownership" relationship with ESTAE routines, and this "ownership" strongly affects whether and at what level user program resumption can occur after an ABEND or breakpoint. Consequently, an understanding of what Z/XDC can and cannot do, and an understanding of how to setup Z/XDC debugging in various environments **requires(!)** an understanding of Request Blocks and how they affect and define your program's environment. For more information, see HELP EXECUTIONLEVELS.

Help DEbugging Amode64

Z/XDC fully supports debugging AMODE-64 programs. When Z/XDC receives control from a program running in 64-bit addressing mode, it of course saves that state and restores it when you use the **GO** or **TRACE** commands to return control to the program.

128-bit PSW

Currently (October 2015), the z/OS Operating System offers only limited support for the execution of code residing "above the bar." (Most System Services [SVCs etc.] cannot be called when execution is above the bar.) However, to the extent that z/OS does allow execution, Z/XDC can be used.

64-bit Registers

A large number of newer machine instructions (regardless of addressing mode) operate on 64-bit wide registers (general purpose registers and control registers), while almost all of the older machine instructions continue to operate on only the lo-order 32 bits of those same registers. So it is useful for Z/XDC to provide ways to reference 64-bit wide registers while retaining an independent way of referencing only the lo-order halves of those registers. Accordingly, Z/XDC uses different names for the different views of the registers:

Rn and **REGS**
These names continue to refer to just the lo-order 32 bits of the 64-bit general registers. **Rn** refers to individual registers (R15, R0, etc.), while **REGS** refers collectively to the entire set of general registers.

RWn and **RWREGS**
These names refer to the **entireties** of the 64-bit general registers. **RWn** refers to individual registers (RW15, RW0, etc.), while **RWREGS** refers collectively to the entire set of 64-bit general registers.

RHn and **RHREGS**
These names refer to the hi-order 32 bits of the 64-bit general registers. **RHn** refers to individual registers (RH15, RH0, etc.), while **RHREGS** refers collectively to the entire set.

Control Registers
A similar arrangement holds for control registers:

CRn and **CREGS:** - These refer to the lo-order 32 bits of the control registers.
CRHn and **CRHREGS:** - These refer to the hi-order 32 bits of the control registers.
CRWn and **CRWREGS:** - These refer to the entire 64 bits of the control registers.

Access Registers
The access registers remain 32 bits wide, so Z/XDC supports only the ARn and AREGS names. There is no need for ARHn or ARWn names.

All of the above described names actually refer to the **retry level** instances of the various registers. To refer to the **error level** instances, simple prefix the appropriate name with the letter **E.** Examples: **ER5 ECRWREGS** etc. For more information, see **HELP EXECUTIONLEVELS.**

64-bit Addressing

Z/XDC’s address expression support has been augmented to support addressing 64-bit

storage and to follow 64-bit wide pointers:

! (64-bit indirect)
When SET BANG 64BIT is in effect (the factory default), the exclamation point can be used in address expressions to load 64-bit wide addresses from the PSW, from 64-bit wide registers or from 64-bit wide locations in storage. For more information, see **HELP ADDRESSING SYNTAX BETWEENTERMS INDIRECT.**

INDIRECT(...)
This built-in function can be used in address expressions for pretty much any kind of indirect operation, including 64-bit indirect operations. For more information, see **HELP FUNCTIONS INDIRECT.**

WIDTH(...)
This built-in function can be used to trim (or zero-pad) an address calculation to a given width. This can be useful for trimming out those pesky AMODE flags sometimes found in the first bit of a 32-bit wide field (or register). For more information, see **HELP FUNCTIONS WIDTH.**

X5(...)
X6(...)
X7(...)
X8(...)
These can be used to extract from storage values to be used within an address expression. The extracted values can be 5 bytes, 6 bytes, 7 bytes or 8 bytes wide. [X1() ... X4() continue to exist.] For more information, see **HELP FUNCTIONS X1-THRU-X8.**

Initial Entry Linkage

When an AMODE(64) program (ie. a program that has been linkedited with the AMODE(64) attribute) first receives control either from the System or from xxxCALL, the registers will be set up as follows:

- **R15:** For AMODE(24) and AMODE(31) programs, R15 points to your program's entry address. For AMODE(64) programs, R15 contains the value **X'FFFFF00x'** (where **x** can be 0, 2 or 4. For more information, refer to IBM's **z/OS MVS Programming: Assembler Services Guide** [SA22-7605], and then search for the string **FFFFF002).**

- **R14:** This register points to a return address that your program should use when it completes. Doing so returns execution to a breakpoint (a DEAD trap, actually) that causes control to pass back to Z/XDC one last time before the debugging session ends. Notes:
  - This return address will always be located in 24-bit storage.
  - If your program does not return via R14 but instead terminates via an SVC 3, then this last breakpoint is bypassed.

- **R13:** This register points to a system provided save area buffer pointed to by the current TCB's **TCBFSA** field. This buffer has the following characteristics:
  - It is 144 bytes long.
  - So it is long enough to contain either a standard, 72-byte register save area, or a 144-byte, F4SA (format-4) register save area.
  - Initially, the 144-byte save area buffer has been entirely zeroed by the system.

- **R1:** This register points either to a TSO-style **CPPL** (Command Processor Parameter List) or to an OS-style PARM field, depending upon whether the debugging session was started by XDCCMD(A) or XDCCALL(A). A **FORMAT R1!** command will show this.

- **R2-R12: !!!WARNING!!!** These registers are **NOT** guaranteed to be zeroed! They may contain random information.

Help DEbugging Batchjobs

Z/XDC has comprehensive support for debugging programs running both in batch job address spaces and in system task address spaces. The **XDCCALL** program can be used to start debugging sessions, and **cdf/XDC** can be used to connect a fullscreen terminal to the debugging session.

The easiest way to start a debugging session in nearly any environment, including in the batch, is to use the XDCCALL program or its XDCCALLA alias. Briefly, XDCCALL[A] can be used as described below. For complete information, see **HELP XDCCALL.**

Another easy way: If a batch job is already running and you want to debug it, but you haven't set it up to run under Z/XDC, you can use a **hooking** capability to start a debugging session "on the fly" in that batch job. In other words, you can start an authorized debugging session in TSO, and from there issue commands that dynamically create a debugging session in your batch job. See **HELP HOOKS** for more information.

XDCCALL (and **XDCCALLA)** provides a high level interface to Z/XDC. It creates a debugging environment within which a user program can be executed. XDCCALL[A] accepts information about a program's name, its parameters if any, its load library(s), and its authorization level, and then it ATTACH's that program as a subtask and establishes Z/XDC as an ESTAI type recovery routine for the subtask. (ESTAI routines are very similar to ESTAEs. They differ only in issues of ownership and in program retry rules. See **HELP EXECUTIONLEVELS** for more information.)

XDCCALL can be used as follows for debugging programs in batch jobs and system tasks:

- **XDCCALL** has three alias names: **XDCCALLA, XDCCMD** and **XDCCMDA,** but you may start your background debugging session using only the XDCCALL or XDCCALLA names. The XDCCMD and XDCCMDA names cannot be used.

- If your program needs to run **authorized,** then use XDCCALLA; otherwise, use XDCCALL.

- Setup your JCL as follows:

  - Change the **EXEC** card as illustrated here:
    From: // EXEC PGM=myprog,PARM='options'
    To: **// EXEC PGM=XDCCALL,PARM='myprog options'**

  - Specifically, move your program's name from the **PGM=** keyword, and insert it, followed by a blank, at the start of the **PARM=** field.

  - Change the **PGM=** keyword to invoke XDCCALL or XDCCALLA instead of your

program.

- If the XDCCALL/XDCCALLA program is not in a **linklist** library, then specify its library in your //JOBLIB or //STEPLIB DD card.

- If the program that you want to debug is not in a linklist library, then specify its library in your //JOBLIB or //STEPLIB DD card or in a **//TASKLIB** DD card.

- HINT: If:
  - You need to run XDCCALLA to debug an **authorized** program,
  - And the program you want to debug is **not located in an authorized library,**
  - Then specify your program's library via **//TASKLIB,** not via //JOBLIB or //STEPLIB.

  See **HELP XDCCALL TASKLIB** for more information.

- If you want to use initial profile settings that are different from Z/XDC's factory defaults, then add an **//XDCPROF** DD card that points to a profile library **(RECFM=FB, LRECL=80, BLKSIZE=n*80).** HINT: This library can be a copy of your ISPF profile library, but don't use the same library; otherwise, you may run into sharing conflicts with your TSO session.

- Include whatever other DD cards your program might need.

Example:

```
//TESTJOB EXEC PGM=XDCCALL,PARM='TESTPROL LINECT=108'
//STEPLIB DD  DISP=SHR,DSN=hlq.XDCV31.XDCLINK
//        DD  DISP=SHR,DSN=hlq.XDCV31.XDCPLPA
//TASKLIB DD  DISP=SHR,DSN=PROL.TESTLIB
//XDCPROF DD  DISP=SHR,DSN=DBCOLE.ISPPROF.COPY2
// <other DD cards needed by TESTPROL>
```

Once you have submitted your job for execution, Z/XDC will almost immediately receive control and then connect to cdf/XDC where it will wait for a programmer to signon to the debugging session. Z/XDC also will notify operations of the pending debugging session by issuing the following WTOs/WTOR:

```
   +DBC640Q xxx v3.1: (jobname) AWAITING USER SIGNON.
    DBC640Q    - Reply C to cancel the wait and percolate the ABEND.
    DBC640Q    - Reply GO to cause the user's program to reattempt execution.
    DBC640Q    - Reply WTOR to issue z/XDC commands via WTO/WTOR.
    DBC640Q    - Reply RESHOW to redisplay these messages.
@nn DBC640Q xxx v3.1: (jobname) Reply C, GO, WTOR, or RESHOW
```

At this point there are two ways to connect to your debugging session:

- Logon to TSO, go into ISPF, find Z/XDC's Startup Panel in ISPF, select Option **3,** and press ENTER. (Ask your Systems Programmer where, in ISPF, he installed the **Z/XDC Startup Panel.)**

- From an idle VTAM terminal, log directly on to cdf/XDC by typing one of the other of the following commands (depending upon VTAM installation settings):
  LOGON APPLID=XDC

LOGON APPLID(XDC)
cdf/XDC will then display a logon screen where you will have to type your TSO userid and password.

Once you have connected to cdf/XDC, you will see a **Job Selection Menu** which should be displaying the name of the job or system task that you want to debug. (If not, then you have security access rule definition problems. See **HELP SECURITY CDF** for more information.) Type **S** next to the jobname. You will now be connected to the debugging session.

When you connect to the session, you will be presented with a fullscreen display with execution interrupted at your program's entry point. The registers will be set to the following standard values:

R15 - If your program starts up in either 24-bit or 31-bit addressing mode, then R15 points to your program's entry address. If your program starts up in 64-bit addressing mode, then R15 contains the value **X'FFFFF00x'** (where **x** can be 0, 2 or 4. For more information, refer to IBM's **z/OS MVS Programming: Assembler Services Guide** [SA22-7605], and then search for the string **FFFFF002).**

R14 - Points to a return address that your program can use to terminate.

R13 - Points to the system provided save area buffer pointed to by the current TCB's **TCBFSA** field. This buffer has the following characteristics:
- It is **144** bytes long.
- So it is long enough to contain either a standard, 72-byte register save area, or a 144-byte, F4SA (format-4) register save area.
- Initially, the 144-byte save area buffer has been entirely zeroed by the system.

R1 - Points to a **PARM** field. The name of your program has been removed. Example: if you specified **// EXEC PGM=XDCCALL,PARM='MYPROG ,,YES',** then the **PARM** field pointed to by R1 will be changed to contain **",,YES".**
- All other registers contain random values. They are **not** pre-zeroed, so your program must not depend on them being zeroed.

PSW - Points to your program's entry point.

At this point, you may want to issue the following typical commands:

- **WHERE**
  This produces a formatted display of the storage where the PSW points (ie. the entry point of your program). Note, at this point, that display probably will be poorly formatted. The next command will take care of that.

- **MAP +0**
  This causes Z/XDC to read at least a module map for your program. This informs Z/XDC where, within your load module, its csects are located. Also, if you have created a csect map for your program, then that will be read as well. See **HELP MAPS** for more information.

  For more information about the **+0** address expression, see **HELP ADDRESSING IMPLICIT.**

- **S Q +0**
  This defines your program's entry csect as a "default" csect. This gives meaning to address expressions that start with a dot. Examples: **.+2C, .TOPLOOP-3B** and just **".".** (See **HELP ADDRESSING PARSERS ASM RESOLUTION** for more information.) Note that the **SET QUALIFIER** command cannot be issued unless a **MAP** command is issued first.

- **WHERE**
  This displays again the code and data located at your program's entry point. If the display still is formatted poorly, then you need to create a csect map to give Z/XDC's formatter better guidance regarding what is data and what is instructions. Alternatively, you can use Z/XDC's **EQUATE** command to define labels with attributes that help to guide the formatter properly. See **HELP COMMANDS EQUATE** for more information.

- You may want to use the **DMAP** and **USING** commands to read and assign to appropriate storage locations dsect maps of IBM control blocks and of your own private control blocks. See **HELP COMMANDS DMAP** for more information.

- At this point you can use Z/XDC's various breakpointing commands to set traps both in the current load module and even in load modules that have not yet been brought into storage (using deferred breakpoints). See **HELP BREAKPOINTS** for more information.

- Finally, use the **GO** command to let your program start executing. When it reaches a breakpoint or an ABEND, Z/XDC will receive control again and present its displays.

Note, under certain conditions, you may have to modify your program's source code in order to use Z/XDC to its fullest capabilities. See **HELP DEBUGGING GETTINGSTARTED** for more information.

Help DEbugging C

c/XDC is a Licensed Feature that provides support for Source Level Debugging for programs written in XL C/C++ and Metal C. c/XDC provides commands, displays and methodologies oriented towards the C programmer. Yet it retains all of the Assembler and z/OS oriented capabilities that have been built up over the decades. So the C programmer will see only displays that make sense to him, yet the bilingual C and Assembler programmer will be able to access the best of both worlds, source displays, object displays, system structures, etc.

Supported C and C++ Compilers

c/XDC supports debugging a wide variety of C languages, including:
- LE C and C++
- Metal C
- XL-clang C and C++
- SPC C (Systems Programming C)
- Dignus Systems/C and C++
- and others

Starting Your Debugging Session

Generally, there are three ways to start a c/XDC debugging session:

- Using the Z/XDC Startup Panel in ISPF to run your program within TSO,
- Running Your Program in the Batch Under XDCCALL[A],
- Using Static or Dynamic Hooks.

These are all discussed in greater detail in HELP DEBUGGING C STARTING.

Source Statement Displays

Once your debugging session is going and you're connected to it, just use the **WHERE** and **FORMAT** commands to display your program. **If:**

- **DWARF Data** has been created for your program, and is findable by c/XDC,
- And if **AUTOSTEP** is in effect,
- And if **AUTOMAP** also is in effect,

Then c/XDC will automatically display your program as C language statements.

At the start of a debugging session, c/XDC will automatically detect if a program is written in XL C/C++ or Metal C. If so, and if **DWARF** data libraries and source program libraries are available, c/XDC will then automatically build a source image map of that program.

It also will automatically advance program execution to the start of the user code (past the C language's prolog code).

So when you start debugging an XL C/C++ or Metal C program, the first display you will see will be your program with execution already advanced to its starting point.

C, C++, and Metal C Program Maps

C program map data is built from something called **DWARF Data** (or simply **DWARF).** Both the XL C/C++ and Metal C compilation processes can create DWARF data. For details, see HELP DEBUGGING C SETUP.

When both **AUTOMAP** and **AUTOSTEP** are set on, c/XDC will automatically load C program maps whenever it detects that execution has reached a new code section.

You can, if you wish, use the **MAP** command to load maps manually. You just have to know the external name of the code that you wish to map. Proceed as follows...

- If a **Binder Map** (i.e. a Linkedit Map) has not yet been built, then use a **MAP modulename** command to build one.
- You can use the **LIST MAPS** command to see what maps have been built.
- Then use the **LIST LKEDMAP modulename CFRIENDLY** command to see a report of all external csect and function names relating to your C/C++ or Metal C code. (The **CFRIENDLY** operand excludes from the report all C support routines and other boilerplate.)
- Decide which code section you want to map, then use an **M** shortcut command at the left to cause the map to be built.

In some cases, especially Metal C cases, c/XDC won't be able to automatically tell where the DWARF Data is, and so you will have to tell c/XDC yourself. There are two

ways to do this:

- The **LIBRARYLISTS** command can be used to build permanent lists of libraries Z/XDC is to search when looking for DWARF Data.
- The **DWARFLIB=** operand on the **MAP** command can be used when you don't want to mess with Library Lists.

Variables, Arrays, Structures, Unions, Whatever

Using information available from DWARF data, c/XDC has full knowledge of almost all of XL C/C++ and Metal C variable types, syntax and formats:

- When displaying XL C/C++ and Metal C variable data, their values are shown according to their types and formats. Use the **LIST VARIABLES** command to display XL C/C++ and Metal C variables.
- When zapping XL C/C++ and Metal C data, you can do so using either XL C/C++ and Metal C formatted data or z/XDC's classic hex and string data formats.
- c/XDC properly handles the displays of arrays, structures, arrays of structures, structures of arrays, unions, ... whatever. c/XDC just does it!
- c/XDC has full knowledge of both XL C/C++ and Metal C variable pools and LE stacks. They all can be displayed by the **LIST VSTACK** command.
- Global constants, subroutine variables, and local variables can always be displayed (and zapped).
- Similarly, variables in pools located in any older, Language Environment Stack Frame also can be displayed/zapped.

The displays produced by the LIST VARIABLES command are in a 3-column tabular format. You can use the **SET VDISPLAY** command to adjust various aspects of the display, such things as gutter widths, indentation, how much of a structure to display, etc. See HELP COMMANDS SET VDISPLAY for details.

Slices

Obviously, when displaying large arrays, it is not feasible to show all array elements, so to help control the overload, the **LIST VARIABLE** command supports a syntax that allows you to:

- Select a list of elements to display,
- Select an n-dimensional subset ("slice") to display,
- Do both simultaneously.

For details, see HELP DEBUGGING C VARIABLES.

Drill Down

When LIST VARIABLES is asked to show a structure, it will show only a single line along with a caption such as:

*** Structure not Explored *** or

*** Structure is Hidden ***
You can use the **L** shortcut to drill down into the structure and to climb back out again.

Parsers

The syntax rules for variables used in XL C/C++ and Metal C differ and conflict with the rules for Assembler. Accordingly, c/XDC introduces the concept of **Parser Management** that allows you, the user, to control:

- Which Parsers (ASM or CEE [or future]) are called for parsing a variable name,
- The order in which the Parsers are called (ASM first or CEE first, etc.),
- And which parser generates the error message should they all fail.

The Parsing Order is displayed by the **LIST PARSEORDER** command and set by the **SET PARSEORDER** command. The setting can be saved in your session profile. For more information, see HELP COMMANDS SET PARSEORDER.

The Parsing Order can also be overridden for individual commands. See HELP ADDRESSING PARSERS TAGS for more information.

Where Are You?

As always, the **WHERE** command can be used to display your program's current execution location. The resulting display will be formatted according to whatever maps have been loaded. If the map is an XL C/C++ or Metal C map, then you will see C source statements. Otherwise, you will see Assembler statements or disassembled machine code.

Breakpoints

For setting breakpoints, just use the **TRAP** and **AT** commands the same as you would for any other program.

Inserting Hooks

Hooks are used for inserting a debugging session into code where a session does not yet exist. (This is as opposed to breakpoints which can be used only once a debugging session already exists.) See HELP HOOKS for more information.

If you wish to insert a dynamic hook into your C program, just use the **HOOK** command just as you would for an Assembler program.

On the other hand, if you wish to compile a static hook into your C program (like what the #XDCHOOK macro does for Assembler programs), your code needs to call a function routine that we provide named **cxdchook.** See HELP DEBUGGING C CXDCHOOK for more information.

Stepping Through Your Code

There is a new command for stepping through your code at the language statement level. It's called **STEP,** and it can be used:

- To step from one statement to the next,
- To step into a subroutine,
- To step out of a subroutine,

For more information, see HELP COMMANDS STEP.

The Underlying Machine Code

Of course, even though c/XDC allows you to display code as if it were High Level Language statements, what's really in storage are machine instructions. So if you are also an Assembler programmer, you can see the underlying instructions simply by using the **BOTH** operand on the **FORMAT** or **WHERE** command, or by using the **SET FORMAT** command to set **BOTH** globally.

You also can use **SET FORMAT CURRENTMCODE** to cause c/XDC to automatically display the underlying machine code for only the C statement that is currently being executed.

Whenever the PSW points to a machine instruction that is within a C statement but is not the first machine instruction underlying that C statement (as can happen when using the **TRACE** command), c/XDC will automatically show that statement's machine code regardless of FORMAT command settings.

And speaking of TRACE, you can still use the **TRACE** command to step through execution at the machine instruction level. In fact, **TRACE** commands and **STEP** commands can be freely intermixed.

Subtopics Index

The following topics provide additional information. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

SETUP - Setting up an XL C/C++ and Metal C debugging session.
STARTING - Starting a debugging session either in TSO or in the batch.
MAPPING - Loading Binder Maps of your program object and DWARF maps of your C source code.
LIBRARYLISTS - Helping the **MAP** command find **DWARF files** and **source files.**
DEBUGGING - Conducting an XL C/C++ and Metal C debugging session.
VARIABLES - Understanding and managing C variable displays.
CXDCHOOK - Compiling a static hook into an XL C/C++ and Metal C program.
CXDCIS - Determining what Z/XDC **clone** to debug with.
EXTERNAL ISSUES - Known issues within c/XDC that are beyond our control.

Only the XL C/C++ and Metal C Languages are Supported

The **c/XDC Feature** supports only those programs that are written using XL C/C++ and Metal C. Other versions of C and C++ are not supported.

Help DEbugging C SEtup

In order to use **c/XDC** to debug your C program, a certain amount of preparation is required, and that prep differs according to whether you are writing in XL C/C++ or Metal C.

XL C/C++ Prep

When compiling XL C/C++ programs, DWARF data (when requested) is built by the XL C/C++ compiler **(CCNDRVR).** When compiling Metal C programs, DWARF data is built by the CDA Assembler **(CDAHLASM).** (See below.)

When compiling an XL C/C++ program in classic **z/OS,** you tell the compiler to produce DWARF data as follows:

- Specify **PARM='DEBUG'.**
- **Do NOT** specify PARM='DEBUG,TEST'. Doing so would cause the compiler to produce something called "ISD-format debugging data", and c/XDC doesn't know anything about that.

When compiling an XL C/C++ program in **USS,** you tell the compiler to produce DWARF data as follows:

- Specify the **-g** switch. This is the equivalent of specifying DEBUG.
- **-g** implies NOOPTIMIZE, so no NOOPTIMIZE parameter needs to be specified.

When the **DEBUG** parm is specified in classic z/OS, the XL C/C++ compiler writes DWARF data to a side file allocated to **//SYSCDBG,** and then the Binder places the dsname of that file into meta-data within the program object. Z/XDC's **MAP** command uses IBM's Common Debug Architecture (CDA) services to find, open and access the DWARF data.

When the **-g** switch is specified in USS, the resulting DWARF file is written to the same folder as the source file. The DWARF file's name will be the same as the source file name but with a **.dbg** extension.

XL C/C++ at Run Time

Then at program run time, you must add **TRAP(ON|OFF,NOSPIE)** to the C Runtime Options part of your job's **PARM** field. Example:

```
//RUN EXEC PGM=XDCCALL,PARM='mypgmname TRAP(ON|OFF,NOSPIE)/'
```

You do not need to add any JCL at run time for referencing the DWARF data file. Z/XDC knows how to find both the DWARF data and the program source files. The DWARF file name is hard-coded within the program object, and the source file names are hard-coded within the DWARF data.

Important! For XL C/C++ programs, in order for the mapping process to be successful, the DWARF data file and all of the needed source program files will need to have the same names as they had when the XL C/C++ program was compiled.

So in the normal case, neither a **LIBRARYLISTS** command nor a SET MAPLIBS command is needed. This is because an XL C/C++ program's program object contains sufficient information for locating that program's DWARF data file without Z/XDC having to be told via a LIBRARYLISTS command. (Also, SET MAPLIBS applies only to ADATA. It is never used for DWARF data.)

However, if you have moved or renamed either the DWARF data files or the source code files, then you can use the **LIBRARYLISTS** commands to override the file information from the program object.

Metal C Prep

For a Metal C program compiled in classic z/OS, the DWARF data is produced by the **CDAHLASM** Assembler and is written to the file allocated to **//SYSDWARF.** It (along with ADATA written to //SYSADATA) is produced automatically (just as long as you do not specify PARM='NODEBUG').

The ADATA can be discarded. Only the DWARF data needs to be kept.

For a Metal C program compiled in USS:

- You need to provide the **-qmetal** option and the **-S** switch on the **xlc** command.
- Then you need to provide the **-g** switch on the **as** command.

The resulting DWARF file is written to the same folder as the source file. The DWARF file's name will be the same as the source file name but with a **.dbg** extension.

No ADATA file is created.

Metal C at Run Time

Unlike with XL C/C++, for Metal C the name of the DWARF data file is **not** saved in the program object or load module. So you will have to use the **LIBRARYLISTS** command in order to make the file known to the MAP command.

Metal C does not have a Language Environment runtime, so there is no need to provide runtime PARMs.

Help DEbugging C STarting

Generally, there are three ways to start a c/XDC debugging session:

- **Using the Z/XDC Startup Panel:** Somewhere in ISPF (you'll have to ask your SysProg where) there is a **Z/XDC Startup Panel** which can be used either to start

a debugging session in TSO or to connect to a debugging session already running in the batch. This panel is discussed in detail in **HELP DEBUGGING ISPF PANEL.** (There also is further information below.)

- **Running Your Program in the Batch Under XDCCALL[A]:** This method allows you to debug your batch job program within its natural environment. The debugging session starts immediately when your program starts. But your program is intercepted at its first machine instruction where it waits for **you** to connect to the session.

- **Using a Static or Dynamic Hook:** A Hook is a mechanism for starting a debugging session at an arbitrary point in an otherwise unprepared program. When execution reaches the hook, a debugging environment is created, and a debugging session starts. But again, everything waits for your connection to the session.

All these methods allow you to debug either authorized (security permitting) or non-authorized programs.

Using the Z/XDC Startup Panel in ISPF

The Startup Panel can be used either to start a debugging session in TSO or to connect to a debugging session already running in the batch (See below). For debugging in TSO, the Panel has numerous fields that need to be filled in. These fields are discussed in detail in **HELP DEBUGGING ISPF PANEL.**

You start a TSO-resident debugging session by using the Panel's **Options 1** (rarely) or **2** (usually). Your program will immediately be loaded into the TSO address space, and your debugging session will commence. For more information, see **HELP DEBUGGING TSO.**

Under the covers, the Panel eventually invokes either XDCCALL or XDCCALLA (for non-authorized and authorized programs, respectively). XDCCALL[A] then builds the environment in which your program will run. For more information, see **HELP XDCCALL.**

The Panel is reasonably self-explanatory but the common things to do are these:

1 - **CMD/PGM Name:** Fill this field in with the name of the program you want to debug.

2 - **Tasklib DSNs:** If the program does not reside in a //STEPLIB library, a linklist library or in common storage (or sometimes even if it does), then you may need to provide the name of the library in which it can be found. This library is then set up as a **tasklib** for your debugging session.

Unfortunately, HFS paths are not currently supported here by the Panel.

3 - **Should CMD/PGM start APF auth?** If you are debugging an authorized program, specify **YES** in this field. This will cause the panel to invoke XDCCALLA instead of XDCCALL.

4 - **Should RENT and REFR pgms be loaded into key 8 Storage?** If you are debugging a **non-authorized** but reentrant program, specify **YES** in this field. This will make it possible to set breakpoints and step through execution.

If you are starting an **authorized** debugging session, then you should almost

always specify **NO** in this field.

5 - **CMD/PGM PARM:** Provide program parms (if any) in this field.

If you are debugging a XL C/C++ program, then you you will also need to specify the **TRAP(OFF,NOSPIE)** LE Runtime parameter in the **CMD/PGM PARM** field. This is needed to disable some of LE's internal recovery routines. See **HELP DEBUGGING LE** for more details.

Metal C programs do not have Language Environment runtime environments, so the **TRAP** parms are not needed in that case.

After you have filled out the Startup Panel, select option **1** (rarely) or **2** (usually) to start your c/XDC debugging session.

For more information about filling out the Panel's various fields, see **HELP DEBUGGING ISPF PANEL.**

Running Your Program in the Batch Under XDCCALL[A]

This method allows you to debug your batch job program within its natural environment. The debugging session starts immediately when your program starts, but your program is intercepted at its first machine instruction and forced to wait until you connect to the session.

To do this, you would change your JCL as follows:

On the **EXEC** card:

- Change the **PGM=** operand to run...
  - XDCCALL for a **non** authorized program,
  - XDCCALLA for an **authorized** program,

- Move the name of the program you want to debug from the PGM= operand to the start of the **PARM=** operand.

- Follow the program name with a **blank** and then **TRAP(OFF,NOSPIE)/**

- If your program has parms of its own, then place those following the **/**

DD cards: There are several session settings that are set by using **//XDCkeywd DD DUMMY** statements. There also are other **//XDCkeywd DD** statements that provide access to needed datasets and libraries. For a more detailed list, see HELP XDCCALL DDNAMES. But here are some of the more common ones to consider:

//TASKLIB DD [loadlibraries]
//STEPLIB DD [loadlibraries]

You can use either or both of these to point to one or more load libraries that your debugging session is going to need. Generally, it doesn't matter which you

use, but you would **use both** if you're debugging an authorized program (via **XDCCALLA)** and your program needs some modules from authorized libraries (use **//TASKLIB)** while others need to come from non-authorized libraries (use **//STEPLIB).** For more information, see **HELP XDCCALL TASKLIB.**

//XDCPROF DD [profilelibrary]
//ISPPROF DD [profilelibrary]
You can use either or both of these to point to FB-80 libraries for storing your Session Profiles.

When **both** are present:
- For PROFILE **READs,** //XDCPROF is checked first, then //ISPPROF is checked second.
- For PROFILE **SAVEs,** //XDCPROF is used, and //ISPPROF is ignored.

If your TSO session allocates its profile library with **DISP=SHR,** then one of these could point to that same library. Otherwise, any suitable FB-80 library will do, just so long as it's a place where you'd want to store multiple Z/XDC Session Profiles.

//XDCRENT8 DD DUMMY
//XDCREFR8 DD DUMMY
You would use both of these if you're debugging a **non-authorized** program (via **XDCCALL)** and your load modules are marked (by the Binder) as being either **RENT** or **REFR.** Normally, z/OS would load such programs into key 0 storage, but when these DDs are present, XDCCALL will cause them to be loaded into key 8 storage, allowing breakpoints to be set and traces to be performed.

Generally, it's a **bad idea** to use these DD statements when debugging authorized programs.

//XDCCMDS DD [sequentialfile]
Use this if you want to run a commands script at session startup time.

For a more complete list of DD statements, see **HELP XDCCALL DDNAMES.**

Connecting to your Debugging Session

You connect to a batch Debugging Session by going to a TSO/ISPF terminal, navigating to the **Z/XDC Startup Panel** and selecting **Option 3.**

When you use **Option 3,** you can **ignore all** of the Panel's fields except the very last (Connect to cdf/XDC using your current TSO Userid?) which you will rarely, if ever, have to change.

When you use **Option 3,** you will be connected to your debugging session already running in the batch. See **HELP XDCSRVER CDF** for more information about the connection mechanism (cdf/XDC).

For more information, see **HELP XDCCALL.**

Using Static or Dynamic Hooks

A Hook is a mechanism for starting a debugging session at an arbitrary point in your program's execution. It can be used when you have not prepared (or do not want to prepare) any special environment for your debugging session. For example, hooks can be used for starting debugging sessions in batch jobs and started tasks where you do not want to change (or cannot change) the JCL to invoke your program within XDCCALL[A].

Broadly speaking, there are two kinds of hooks:

- **Static Hooks** are hooks that you compile into your program. For C programmers, we provide a function named **cxdchook** that your program can call for starting a debugging session. This function is described in **HELP DEBUGGING C CXDCHOOK.**

- **Dynamic Hooks** are hooks that are created during execution by the **HOOK** command. They can be set on the fly into programs and address spaces that are already running. Dynamic Hooks can be set:
  - Into load modules and program objects that are already present in the current aspace.
  - Into load modules and program objects that will be loaded in the future into the current aspace.
  - Into load modules and program objects that are already present in any **other** address space (security and authority permitting, of course).

Important! This last item is a very powerful tool! With it, you can start an ad-hoc debugging session in any running address space that is misbehaving or that you'd just like to investigate. For more information, see HELP HOOKS DYNAMIC OTHERSPACES STARTINGNEWSESSIONS.

Again, you connect to the resulting debugging session by going to a TSO/ISPF terminal, navigating to Z/XDC's Startup panel, and selecting **Option 3** (see above).

For more information, see **HELP HOOKS.**

Help DEbugging C Mapping

c/XDC builds C, C++ and Metal C Source Code Maps from DWARF Data. DWARF Data is created by the C program compilation process. For details on making this happen for both C/C++ and Metal C, see HELP DEBUGGING C SETUP.

C Maps can be loaded either automatically or manually:

- When **AUTOMAP** is turned on, maps can be loaded automatically.
- Use the **MAP** command to load maps manually.

Binder Maps and C Maps

For the purposes of this discussion, there are two kinds of maps: (Dsect maps are discussed elsewhere):

- **Binder Maps** (also known as Module Maps or Linkedit Maps) are maps of the load module or program object that contains your program. They inform Z/XDC of where, within a load module (or program object), the program's csects are.

  Z/XDC builds Binder Maps from a module's ESD (External Symbols Dictionary) data. Before c/XDC can build a Source Code Map, it must first build a Binder Map. This is something that Z/XDC does automatically when it needs to.

- **Csect** maps contain information about the source code inside a code section (csect). These maps come in three flavors:
  - **SYM Data** maps are symbol maps for programs written in Assembler.
  - **ADATA** maps are source image maps also for programs written in Assembler.
  - **DWARF Data** maps are source image maps for programs written in C, C++, Metal C or Systems/C.

You can use the **LIST MAPS** command to see what maps currently are built and what their types are.

AUTOMAPing

When AUTOMAPing is turned on, c/XDC will automatically attempt to build a csect map whenever it sees that execution has reached a previously unmapped code section (csect).

Z/XDC will build maps both for Assembler and for C, C++, Metal C and Systems/C programs. The maps will be built only when the required data (DWARF Data, ADATA or SYM Data) exists and Z/XDC knows where to find it.

When execution reaches a previously unmapped load module or program object, Z/XDC will also automatically build a Binder Map before building the appropriate code section map (csect map).

To manage AUTOMAPing, use the following commands:

SET AUTOMAP ON|OFF	- This turns AUTOMAPing on or off.
LIST AUTOMAP	- This reports the current setting for AUTOMAP.
PROFILE SAVE [name]	- The AUTOMAP setting is part of your session profile. This command saves your profile to disk for reloading on another day.

AUTOMAPing works best when **AUTOSTEP'ing** is also turned on. AUTOSTEP'ing is what causes c/XDC to allow a program's execution to automatically proceed from a module's Entry Point, through all Prolog Code, and eventually stopping at the start of a C, C++ or Metal C's first language statement.

To manage AUTOSTEP'ing, use the following commands:

SET STEP AUTOSTEP=ON|OFF	- This turns AUTOSTEP'ing on or off.
LIST STEP	- This reports the current setting for AUTOSTEP.
PROFILE SAVE [name]	- The AUTOSTEP setting is part of your session profile.

This command saves your profile for reloading on another day.

The default Factory Reset profiles, **C80** and **CWIDE,** have both AUTOMAP and AUTOSTEP turned on.

Manual MAP'ing

If AUTOMAP is turned off, or if you wish to map a code section that has not been visited yet (or more accurately that Z/XDC does not know has been visited yet), then you can build the maps manually using the **MAP** command. Here are some typical examples:

MAP modulename.csectname
or **MAP modulename.functionname**
(Syntactically, these two forms are equivalent.) If you know the external name of the code section or function you want to map, then this form of the MAP command will do the trick.

1) **MAP modulename**
2) **LIST LKEDMAP modulename CFRIENDLY**
3) **MAP modulename.externalname**
4) **SET QUALIFIER modulename.externalname**
5) **FORMAT .#nnn** [display via statement number]
6) **FORMAT .+hhhhh** [display via offset]

Sometimes, you want to load a C map, but you don't know the external name of the routine you want to map. Maybe you forgot it. Maybe the external name is a mangling of the function name, whatever. So here's what to do:

1) **MAP modulename**
First, build just the Binder Map.

2) **LIST LKEDMAP modulename CFRIENDLY**
Then use the C-Friendly version of the LIST LKEDMAP to display the pared down version of the module's Binder Map. All of the LE C related boilerplate is filtered out, leaving only those external names that arise directly from your C (or Assembler) code.

3) **MAP modulename.externalname**
Now, examine the map to find the external name of the code you want to map. Then issue the MAP command a second time to load the map.

4) **SET QUALIFIER modulename.externalname**
(optional) Use the SET QUALIFIER command to set up the module name and external name as defaults. This allows them to be represented by a single, leading dot (.) in subsequent address expressions.

5) **FORMAT .#nnn** [display via statement number]
Example: **FORMAT .#38** shows statement #38 in the default code section.

6) **FORMAT .+hhhhh** [display via offset]
Example: **FORMAT .+3DE0** shows whatever's at +X'3DE0' past the start of the default modulename.externalname.

Note, all of the above commands have **shortcuts.** See **HELP SHORTCUTCMDS** for details.

Help DEbugging C Librarylists

For a Glossary of Terms, see **HELP DEBUGGING C LIBRARYLISTS GLOSSARY.**

When the c/XDC Licensed Feature is installed, the **MAP** command can be used to build Source Image Maps of XL C/C++ and Metal C programs and to map their variable pools. In order to do this, c/XDC needs access to two things:

- **DWARF data file:** This is a file that is produced by the C program compilation process. It contains the information necessary for mapping variable pools and finding C source code files. For more information, see **HELP * DWARF.**

- **C source code files:** These are the one or more source code files that were compiled to produce your C program. For more information, see **HELP * CSOURCE.**

Problems occur, of course, when the DWARF and/or C source file **names** either are not present in your C program's metadata or have been changed since compile time. This means that if c/XDC is to succeed in building its maps, it's going to have to have a method for determining a DWARF file's or C source code file's **current** dataset name or file name. This correction process is called **redirection.** For more information, see **HELP * REDIRECTION.**

The file name correction process is accomplished through the creation of lists that connect Match Patterns to candidate substitute file names. So when c/XDC has a file name that needs correcting, it calls an internal service we call **Library Lists Management.** LLM creates and manages lists of Match Patterns connected to lists of candidate file names. For more information about these lists, see **HELP * LISTS.**

Another mechanism for providing corrected file names to c/XDC is to use **DWARFLIB=** and **SOURCELIB=** operands on the **MAP** command. See **HELP * SEARCHORDER** for more information.

For information about managing the Library Lists, see **HELP * MANAGEMENT.**

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

DWARF - Accessing DWARF data for building maps

- **CSOURCE** - Accessing C source files for building maps
- **REDIRECTION** - Redirecting file names from C program metadata to a DWARF file's or C source file's actual name
- **LISTS** - Creating lists of file names and name matching patterns
- **SEARCHORDER** - Places where c/XDC checks when resolving DWARF and C source file names
- **MANAGEMENT** - Creating, changing displaying and deleting library lists
- **GLOSSARY** - Definitions of terms pertaining to Library Lists Management

Help DEbugging C Librarylists Dwarf

For a Glossary of Terms, see **HELP DEBUGGING C LIBRARYLISTS GLOSSARY.**

DWARF Data Side Files

When an XL C/C++ or Metal C program is compiled, one option is to produce debugging information in the form of a **DWARF** data side file. The method differs between XL C/C++ vs. Metal C, so see **HELP DEBUGGING C SETUP** for details.

DWARF is a rather strained acronym that stands for "Debugging With Attributed Record Formats". It is an industry standard format used by some compilers for conveying debugging information to tools such as c/XDC. In recent years, IBM has updated its XL C/C++ and Metal C compilers to use the DWARF format for its debugging data.

Support in Z/XDC for using DWARF data is part of the **c/XDC Licensed Feature.** For more information about Licensed Features, see **HELP SUPPORT FEATURESANDCAPS.**

When DWARF data is produced for an **XL C/C++** program, the dsname or path/filename of the DWARF side file winds up being hardcoded into metadata associated with the resulting XL C/C++ compilation unit (also referred to as a code section or csect). This hardcoded dsname or filename is then propagated forward by the Binder into the resulting program object.

On the other hand, when DWARF data is produced for a **Metal C** program, the name of the DWARF side file is lost. You, the programmer, will have to know what it is and where it is, and you will have to use the **LIBRARYLISTS** command to tell c/XDC where it is.

Problems with Finding DWARF Files

Problems occur, of course, when the DWARF file **name** either is not present or has been changed since compile time. This can arise when any of the following occur:

- **The program is a Metal C program:** The DWARF side file's dsname or path/filename is not recorded in the program object or load module.

- **The DWARF data side file has been renamed or moved into a different library or folder:** The recorded dsname or path/filename no longer is correct.

Finding the Correct Files

There are two ways to tell c/XDC where the DWARF date file really is:

- The **MAP** command has a **DWARFLIB=** operand that you can use to tell c/XDC exactly where to look.

- Or the **LIBRARYLISTS** command can be used to create, activate and generally manage lists of PDS[E] libraries, sequential files and HFS/ZFS folders/files, and //ddnames that c/XDC is to search for DWARF data.

For more information, see **HELP *UP REDIRECTION.**

Or use **HELP *NEXT (F11)** to continue on to the next topic.

Help DEbugging C Librarylists Csource

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

C Source Code Files

In order to build maps of C programs, c/XDC needs, not just DWARF files, but also **C source files** too. Fortunately, the names of those files are stashed in the DWARF, so c/XDC starts by looking there.

The DWARF data contains hardcodings of the dsnames and/or path/filenames of all the C source files involved in the compilation. This is true both for DWARF created by the XL C/C++ compiler and for DWARF created by Metal C's CDAHLASM post-compile Assembler.

Problems with Finding C Source Files

Problems occur, of course, when the C source file names either are not present or have been changed since compile time. This can arise when any of the following occur:

- **One or more of the C source files have been renamed or moved into different datasets or files:** The recorded names no longer are correct.

- **The C program's source statements were coded inline with the JCL** (i.e. following a **//SYSIN DD *** statement): The recorded C source dsname is useless. (If you have a copy of the inline code lying around somewhere, you can use the **LIBRARYLISTS** commands to redirect c/XDC to use that instead.)

Finding the Correct Files

There are two ways to tell c/XDC where the source program files really are:

- The **MAP** command has a **SOURCELIB=** operand that you can use to tell c/XDC exactly where the source code library is.

- Or the **LIBRARYLISTS** command can be used to create, activate and generally manage lists of PDS[E] libraries, sequential files and HFS/ZFS folders/files, and //ddnames that c/XDC is to search for the source files.

For more information, see HELP *UP REDIRECTION.

Help DEbugging C Librarylists Redirection

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

When the c/XDC Licensed Feature is installed, the **MAP** command can be used to build Source Image Maps of XL C/C++ and Metal C programs and to map their variable pools. In order to do this, c/XDC needs access to two things:

- **DWARF data file:** This is a file that is produced by the C program compilation process. it contains the information necessary for mapping variable pools and finding C source code files. For more information, see HELP * DWARF.

- **C source code files:** These are the one or more source code files that were compiled to produce your C program. For more information, see HELP * CSOURCE.

Problems with Finding DWARF and C Source Files

Problems occur, of course, when the DWARF and/or C source file **names** either are not present or have been changed since compile time. This can arise when any of the following occur:

- **The program is a Metal C program:** The DWARF side file's dsname or path/filename is not recorded in the program object or load module.

- **The DWARF data side file has been renamed or moved into a different library or folder:** The recorded dsname or path/filename no longer is correct.

- **One or more of the C source files have been renamed or moved into different datasets or files:** The recorded names no longer are correct.

- **The C program's source statements were coded inline with the JCL** (i.e. following a **//SYSIN DD** * statement): The recorded C source dsname is useless.

This means that if c/XDC is to succeed in building its maps, it's going to have to

have a method for determining a DWARF file's or C source code file's **correct** dataset name or file name. This correction process is called **Redirection.**

The LIBRARYLISTS Command

This is where **Library Lists** come into play:

- **Library Lists** are lists of libraries, folders, files, datasets and //ddnames in which c/XDC will search for a needed DWARF data file name or C source file name.

- The **LIBRARYLISTS** command allow you to create and manage and save these lists.

The Redirection Process

When c/XDC obtains a proposed dataset/file name from the program object, the Library Lists can be searched to see if the provided name needs to be corrected. If the name matches a **Redirect List** entry's Match Pattern, one or more candidate replacement names are built and returned one by one to c/XDC for its consideration.

This continues either until c/XDC finds the DWARF data or C source code it is looking for, or until Library Lists Management runs out of candidate names to pass back to c/XDC.

For more information, see HELP *UP LISTS.

Help DEbugging C Librarylists Lists

For a Glossary of Terms, see **HELP DEBUGGING C LIBRARYLISTS GLOSSARY.**

Library Lists

The **LIBRARYLISTS** command allows you to create **named lists** of candidate libraries, folders, files, datasets and //ddnames in which c/XDC will search for a needed DWARF data file name or C source file name.

Three Types of Lists

The **LIBRARYLISTS** command builds and manages three kinds of lists. They are:

- A **Redirect List:** This is a single list that consists of one or more entries that connect Match Patterns to Lists of candidate files, folders, datasets, libraries and //ddnames that c/XDC is to search for the needed DWARF data or C source file. For more information, see **HELP * REDIRECTLIST.**

- **Candidates Lists:** These are lists of files and libraries that are pointed to by Redirect Lists. When an Original File Name is matched by a Redirect List's Match

Pattern, the associated Candidates List is appended to a Results List. For more information, see **HELP * CANDIDATESLISTS.**

- **Results List:** This is a temporary list that Library Lists Management itself builds when resolving an Original File Name obtained from a C program's metadata. Whenever such a file name matches a Pattern in a Redirect List, that list's Candidates List is appended to the Results List. Eventually, the files from the Results List are searched one by one for the DWARF data or Source code that is needed by the MAP command.

  When LLM receives an Original Name from a C program, it scans the Redirect List for **all** entries whose Match Patterns match the given name. LLM then builds a Results List from a concatenation of all the Candidates Lists linked to the matched Redirect List entries. Thus, the Results List is a list of datasets, files and libraries that might contain the desired DWARF data or C source code.

  Then LLM passes each Results List entry, one by one, back to c/XDC for examination. c/XDC searches the given dataset or library to see if it contains the needed DWARF or C source. If so, then the data is read and used to build whatever map is being built. But if not, then c/XDC asks LLM to return the next entry in the Results List. This continues until either a map is built or the List is exhausted.

DWARF Lists vs. CSOURCE Lists

Library Lists Management allows you to create two separate **Families** of Library Lists, one for finding DWARF data files and one for finding C source files. The two Families are named:
- **DWARF** for finding DWARF data files,
- **CSOURCE** for finding C source files.

For more information, see **HELP * FAMILIES.**

Saving and Restoring Library Lists Data

Library Lists Data can be permanently stored into your Session Profile. From there, it can be reloaded both at will and at the beginning of your Debugging Sessions.

For more information, see **HELP * SAVING.**

Help DEbugging C Librarylists Lists REDirectlist

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

A **Redirect List** is a single list that consists of one or more entries that connect Match Patterns to named Candidates Lists consisting of the names of files, folders, datasets, libraries and //ddnames that c/XDC is to search for the needed DWARF data or C source file.

Or if you have only one substitute name for your Original Names, then you can dispense with creating a single-entry Candidates List and instead link the Match Pattern directly to that single name.

Library Lists Management maintains **two Families** of lists, one for **DWARF data** searches and one for **C source code** searches. Each Family contains one Redirect List and zero or more named Candidates Lists. For more information, see HELP *UP FAMILIES.

When c/XDC calls Library Lists Management, it passes the **Original Name** of a dataset or file that is supposed to contain DWARF data or C source code. LLM then searches the appropriate Family's Redirect List looking for one or more entries that match the Original Name. It then uses the matched entries to build a **Results List** of one or more **Candidate Names** of datasets, files, libraries and folders. These names then get passed back to c/XDC for its consideration.

Redirect List Entries

Each entry in a Family's Redirect List contains two fields:

- The first is a **Match Pattern** containing wildcard characters (? * and **) intermixed amongst constant characters. A Match Pattern is intended to match one or more Original Names from c/XDC. (Example: **MY.LIBRARY(*)**). For more information about wildcard characters, see HELP * WILDCARDS.

- The second field is one or more **Result Objects.** These objects may be the names of datasets, files, folders, libraries, //ddnames and named **Candidates Lists** to be further examined, resolved and returned. For detailed descriptions of Result Objects, see HELP *UP CANDIDATESLISTS RESULTOBJECTS.

When an Original Name is received from c/XDC, that name is used to search the appropriate Family's Redirect List. For each matched entry, one or more Candidate Names is added to the Results List.

The Result Objects

Result Objects are used in the construction of a **Results List.**

A **Results List** is a list of one or more datasets, files, folders and libraries that c/XDC is to search in order to find the DWARF data or C Source code that it needs for building its map. For more information, see HELP *UP RESULTSLIST.

A **Result Object** can be any of the following:
- The name of a classic z/OS sequential file
- (Including a specific PDS member name)
- The name of a library (PDS or PDSE)
- The name of an HFS file (including its path)
- The name of an HFS folder
- The ddname of a current allocation of one or more datasets, files, folders and/or libraries
- The name of a Candidates List (to be included by reference)

For detailed information, see HELP *UP CANDIDATESLISTS RESULTOBJECTS

Match Variables

A **Match Variable** is a variable whose value is set during the Pattern Match process. When an Original Name is received from c/XDC, that name is tested against each Match Pattern in the Redirect List. When a match occurs, each consecutive string of wildcards in the pattern gives rise to one Match Variable. The value of that variable is the exact part of the Original Name that was matched by the wildcard string. For more information, see HELP * MATCHVARIABLES.

Once the Results List is built, LLM passes its entries one by one, back to c/XDC for its consideration.

Some Tools to Test With

To help you to understand **Match Variables** better, try using the **LIBRARYLISTS TEST MATCH** command. For a given **Original Name** and a given **Match Pattern,** this command will display exactly the values the Match Variables are set to. You can use this command **before** you build your lists. For more information, see HELP COMMANDS LIBRARYLISTS TEST MATCH.

Once you've built your **Redirect List** and your various **Candidates Lists,** you might want to run a test or two to see how they all work. If that's the case then the **LIBRARYLISTS TEST TRY** command is a handy command to know. You can give it an **Original Name** to try out, and it will display the **Results Lists** that would be built from the Lists that you have created. For more information, see HELP COMMANDS LIBRARYLISTS TEST TRY.

Help DEbugging C Librarylists Lists REDirectlist Wildcards

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

Wildcard Characters

Redirect List Match Patterns may contain the following **Wildcard Characters:**

- **?** matches exactly one character,
- * matches 0 or more characters up to the next period, open parenthesis, close parenthesis or forward slash [**.** **)** **(** **/**].
- ** matches 0 or more characters up to the end of the Original Name from c/XDC.

All wildcards can be following by other characters, both wild and otherwise.

The * and ** wildcards are **greedy.** A leftwards instance will match the longest string that does not cause the overall match to fail.

When a match occurs, each string of consecutive Wildcard Characters gives rise to one **Match Variable** which can be used in the **Individual Result Objects** to create **Candidate Names** that are reactive to the content of the **Original Name** from

c/XDC. For more information, see HELP ∗UP MATCHVARIABLES.

Examples

Here are some examples of Match Patterns with explanations of what they do.

∗.A???B∗∗C?∗DDD(∗)
This might be used to match certain classic z/OS library names:

- The first ∗ matches any single high level qualifier (such as a userid).
- The **???** matches exactly three characters. They can be any three characters, including periods (.).
- The ∗∗ matches any zero or more characters (including periods) that do not cause the rest of the Match Pattern to fail.
- The **?∗** matches a string of **one** or more characters that do **not** contain periods and that do not cause the rest of the Match Pattern to fail. In this case, the longest string that ?∗ can match is 4 characters.
- The final ∗ matches any library member name.

∗∗(∗)
This might be used to match **any** classic z/OS library name and any member name within that library.

∗∗/∗
This might be used to match **any** HFS path to any file name.

Help DEbugging C Librarylists Lists REDirectlist Matchvariables

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

When c/XDC calls Library Lists Management, it passes the **Original Name** of a dataset or file that is supposed to contain DWARF data or C source code. LLM then searches the appropriate Family's **Redirect List** looking for one or more entries that match the Original Name. It then uses the matched entries to build a Results List of one or more **Candidate Names** of datasets, files, libraries and folders. These names then get passed back to c/XDC for its consideration. For more information, see HELP ∗UP.

LLM searches a Redirect List by comparing the Original Name from c/XDC to each List entry's Match Pattern. A Match Pattern may contain Wildcard Characters [**?** ∗ ∗∗] allowing the Pattern to match a variety of Original Names. For more information

about wildcards, see HELP *UP WILDCARDS.

Match Variables

Individual Result Objects may contain **Match Variables** that represent portions of the Original Name from c/XDC that were matched by Wildcards in the Match Pattern. This allows parts of the Original Name to be carried into the Candidate Name. This is very useful for Candidate Names containing such things as userids, project names, member names and the like.

The Pattern Match process creates Match Variables as follows:

- The created Variables are very temporary. They exist only in the context of the Redirect List entry currently being examined by Library Lists Management.

- Each string of consecutive Wildcards gives rise to one Match Variable whose value will be the specific characters matched by that Wildcard string.
  Example: ***.A???B**C?*DDD(*)** This generates five match variables:
  - The first is whatever is matched by the first *
  - The second is whatever is matched by **???**
  - The third is whatever is matched by **
  - The fourth is whatever is matched by **?***
  - The fifth is whatever is matched by the last *

- The names of the generated match variables are ampersands (**&**) followed by the sequence number of the wildcard string. The sequence number may range from **1** to **9.**

  So continuing the above example:
  - **&1** is whatever was matched by the first *
  - **&2** is whatever was matched by **???**
  - **&3** is whatever was matched by **
  - **&4** is whatever was matched by **?***
  - **&5** is whatever was matched by the last *
  - **&6** thru **&9** are set to null.

- The matched text can be either upcased or locased by using the names:
  - **&U1** thru **&U9** (for upcasing)
  - **&L1** thru **&L9** (for locasing)

  The case of the variable names doesn’t matter. For example, **&U2** and **&u2** both reference an upcasing of the text matched by **&2.**

The match algorithm is **greedy.** This means that a wildcard string to the left will match the longest string that doesn’t fail the match as a whole.

A Tool to Test With

You can use the **LIBRARYLISTS TEST MATCH** command to see exactly how Match Variables get set for a trial **Original Name** matched against a trial **Match Pattern.** See HELP COMMANDS LIBRARYLISTS TEST MATCH for details. (There’s also an example below.)

Examples

Continuing the above example [***.A???B**C?*DDD(*)**] suppose that c/XDC called Library Lists Management with an Original Name of:
DBCOLE.AUSEBNOT.MYDEPT.MYPROJ.MCCCCDDD(MEMBER). Then:

- **&1** would be **DBCOLE**
- **&2** would be **USE**
- **&3** would be **NOT.MYDEPT.MYPROJ.MYCCC**
- **&4** would be **C**
- **&5** would be **MEMBER**
- **&6** would be null

Notice that a successful match could have worked out if **&3** had been matched by Library Lists Management to any of the following:

```
    &3 then &4 would have been:
    NOT.MYDEPT.MYPROJ.MYCCC  C
    NOT.MYDEPT.MYPROJ.MYCC   CC
    NOT.MYDEPT.MYPROJ.MYC    CCC
```

However, the match that Library Lists Management chose was the one that made the left most of the two variables (i.e. the **&3** variable) the longest it could be and still have a successful match. (That's the match shown first in the above table.)

You can use the following command to see all this in action:

```
    LIBRARYLISTS TEST MATCH    ...
    PATTERN=*.A???B**C?*DDD(*) ...
    FILE=DBCOLE.AUSEBNOT.MYDEPT.MYPROJ.MCCCCDDD(MEMBER)
```

Or more compactly:

```
    LL TES MAT PAT=*.A???B**C?*DDD(*) ...
    F=DBCOLE.AUSEBNOT.MYDEPT.MYPROJ.MCCCCDDD(MEMBER)
```

Either way, the result is this:

```
   Testing DBCOLE.AUSEBNOT.MYDEPT.MYPROJ.MCCCCDDD(MEMBER)
   Against pattern *.A???B**C?*DDD(*)
   Pattern result is MATCH

   &1="DBCOLE"
   &2="USE"
   &3="NOT.MYDEPT.MYPROJ.MCC"
   &4="C"
   &5="MEMBER"
   &6=""
   &7=""
   &8=""
   &9=""
```

Help DEbugging C Librarylists Lists Candidateslists

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

A **Candidates List** is a list of **Result Objects** that are used to build a **Results List** that will be returned to c/XDC. **Result Objects** are described in detail in HELP * RESULTOBJECTS.

Result Lists are described in detail in HELP *UP RESULTSLIST.

Candidates Lists are always named. There can be any number of Named Candidates Lists.

A Named Candidates List can be linked to either by a **Redirect Lists's** entry or by entries in other Candidates Lists.

When an **Original Name** (from c/XDC) is matched against a suitable Family's Redirect List entry, if that entry's Result Object is the name of a Candidates List, then that list is resolved, and its entries are added to the **Result's List.**

If a Candidates List itself contains entries that link to **other** Candidates Lists, then those lists too are incorporated into the resolution process.

For more information, see:
    HELP *UP REDIRECTLIST
    HELP *UP RESULTSLIST

The Result Objects

All entries in a Candidates List are **Result Objects.** Result Objects are used in the construction of a **Results List.**

A **Results List** is a list of one or more datasets, files, folders and libraries that c/XDC is to search in order to find the DWARF data or C Source code that it needs to build its map. For more information, see HELP *UP RESULTSLIST.

A **Result Object** can be any of the following:
- The name of a classic z/OS sequential file
- (Including a specific PDS member name)
- The name of a library (PDS or PDSE)
- The name of an HFS file (including its path)
- The name of an HFS folder
- The ddname of a current allocation of one or more datasets, files, folders and/or libraries
- The name of another Candidates List (to be included by reference)

For detailed information, see HELP * RESULTOBJECTS

Match Variables

A **Match Variable** is a variable whose value is set during the Pattern Match process. When an Original Name is received from c/XDC, that name is tested against each Match Pattern in the Redirect List. When a match occurs, each consecutive string of wildcards in the pattern gives rise to one Match Variable. The value of that variable is the exact part of the Original Name that was matched by a wildcard

string. For more information, see HELP *UP REDIRECTLIST MATCHVARIABLES.

Once the Results List is built, LLM passes its entries one by one, back to c/XDC for its consideration.

Help DEbugging C Librarylists Lists Candidateslists Resultobjects

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

Result Objects

All entries in a Candidates List are **Result Objects.** All entries in a Redirect List are Match Patterns Linked to Result Objects. Result Objects ultimately are used in the construction of a **Results List.** For information about Redirect Lists and Candidate Lists, see:
- HELP DEBUGGING C LIBRARYLISTS LISTS REDIRECTLIST
- HELP DEBUGGING C LIBRARYLISTS LISTS CANDIDATESLISTS

A **Results List** is a list of one or more datasets, files, folders and libraries that c/XDC is to search in order to find the DWARF data or C Source code that it needs to build its map. For more information, see HELP *UP *UP RESULTSLIST.

A **Result Object** can be any of the following:
- The name of a classic z/OS sequential file
- (Including a specific PDS member name)
- The name of a library (PDS or PDSE)
- The name of an HFS file (including its path)
- The name of an HFS folder
- The ddname of a current allocation of one or more datasets, files, folders and/or libraries
- The name of another Candidates List (to be included by reference)

Result Objects may be either **individual** or **collective:**

- An **individual** Result Object is almost ready to be added directly to the Results List that Library Lists Management is building. All that's needed is for **Match Variables** (if any) to be resolved. Once resolved, the Result Object becomes a **Candidate Name** and is added to the Results List.

- A **Collective** Result Object is a name that represents **multiple** Result Objects. In this case, the Collective Result Object will eventually be resolved into multiple Individual Result Objects that will be further resolved into Candidate Names and added to the Results List.

Individual Result Objects

Individual Result Objects are strings that may contain **Match Variables.** But once those Variables are resolved, the strings become **Candidate Names** that are ready to be added to a **Results List.** Candidate Names may be any of the following:
- It may be the name of a **classic z/OS** sequential file.

- It may be the name of a specific library member [e.g. pdsname(member)].
- It may be the name of an entire PDS or PDSE library (without a member name specified).
- It may be the path and filename of an **HFS** file.
- It may be the path and name of an **HFS** folder.

In all of these cases, the dataset name, library name, file name or folder name is added to the Results List.

Individual Result Objects may contain **Match Variables.** These are variables whose values are set during the Pattern Match process from those portions of the Original Name that were matched by Wildcard Character strings. Match Variables in the Result Objects will be replaced by their values and then added to the Results List as Candidate Names. See HELP DEBUGGING C LIBRARYLISTS LISTS REDIRECTLIST MATCHVARIABLES for more details.

Collective Result Objects

A Collective Result Objects may be any of the following:

- It may be the **ddname** of any current allocation of one or more (i.e. a concatenation of) datasets, libraries, files and folders.

  If the allocation is for a a concatenation, then **each** file (etc.) is treated as an Individual Candidate Name, and is added to the Results List.

- A Collective Result Object may also be the name of a **Candidates List,** in which case:
  - Each **Individual** Object in the List is resolved to a Candidate Name and added to the Results List,
  - While each **Collective** Object in the List is further resolved.

Collective Result Objects may **not** contain Match Variables, but the Individual Objects they are resolved into may.

The Syntax of Result Objects

When the name of a Result Object is provided on a LIBRARYLISTS command, the type of object that it is is determined partially by Library Lists Management and partially by c/XDC upon receiving a Candidate name from LLM. The determination process is as follows:

Recognizing DDnames

LLM first checks to see if the string is a ddname. The string is recognized as being a **ddname** when the string starts with a pair of slashes (//) followed by up to 8 characters that conform to the standard syntax rules of ddnames. Example: **//DWARFLIB**

Recognizing HFS Files and Folders

Otherwise, LLM checks to see if the string contains one or more forward slashes (/). If so then it is recognized as being an HFS path to a file or folder. Further determination is done by c/XDC when it receives the string from LLM. Examples:

- **/u/dbc/dwarflib/** [a folder, processed like a library]
- **/u/dbc/dwarflib/myceepgm** [a file, processed sequentially]

Recognizing Classic z/OS Datasets and Libraries

Otherwise, LLM checks to see if the string contains one or more periods (.). If so then it is recognized as being a classic z/OS dataset name, PDS[E] member name or PDS[E] library. (Further determination is done by c/XDC when it receives the string from LLM.) Examples:

- **DBCOLE.MYPROJ.DWARFLIB** [a library]
- **DBCOLE.MYPROJ.DWARF.MYCEEPGM** [a sequential file]
- **DBCOLE.MYPROJ.DWARFLIB(MYCEEPGM)** [a specific PDS member, processed sequentially]

Recognizing Candidates Lists

Otherwise, LLM checks to see if the string consists entirely of up to 8 alphabetics and/or digits. If so then it is recognized as being a Candidates List name (which would require further resolution).

Note, it is perfectly fine if the Candidates Lists's name is a pure number.

Recognizing Leftovers

Otherwise, LLM throws up its hands and says, "Well this must be an HFS name after all."

Help DEbugging C Librarylists Lists RESultslist

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

A **Results List** is a temporary list that Library Lists Management builds when resolving an Original Name that it has received from c/XDC.

When LLM receives an **Original Name,** it scans the Redirect List looking for **all** entries whose Match Patterns match the given name. LLM then builds a Results List from a concatenation of all the Individual Result Objects found in or implied by the **Candidates Lists** linked to the matched Redirect List entries.

Before an Individual Result Object is added to the Results List, if it contains any **Match Variables,** then those are resolved with the values they received from the **Match Pattern** to which the Candidates List is directly or indirectly linked. See HELP *UP REDIRECTLIST MATCHVARIABLES for more information.

Note, this resolution process of Match Variables is what converts an Individual Result Object into a Candidate Name, ready to be added to the Result List.

When the Results List is completed, its entries are passed one by one back to c/XDC for its consideration. If c/XDC rejects one Candidate Name, it returns to Library Lists Management to receive the next.

Testing Results Lists

Once you've built your **Redirect List** and your various **Candidates Lists,** you might want to run a test or two to see how they all work. If that's the case then the **LIBRARYLISTS TEST TRY** command is your friend. You can give it an **Original Name** to try out, and it will display the **Results Lists** that would be built from the Lists that you have created. For more information, see HELP COMMANDS LIBRARYLISTS TEST TRY.

Help DEbugging C Librarylists Lists Families

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

DWARF Lists vs. CSOURCE Lists

Library Lists Management allows you to create two separate **Families** of Library Lists, one for finding DWARF data files and one for finding C source files. The two Families are named:

- **DWARF** for finding DWARF data files,
- **CSOURCE** for finding C source files.

Each Family has its own anchoring **Redirect List** which may contain:

- **Individual Result Objects** which link directly to individual datasets, files, folders and libraries,
- **Collective Result Objects** which may link to any one or more //ddnames or existing **Candidates Lists.**

So Candidates List can be created that contain only DWARF file and library names, while others may contain only C source code names.

Help DEbugging C Librarylists Lists Saving

For a Glossary of Terms, see **HELP DEBUGGING C LIBRARYLISTS GLOSSARY.**

Saving Library Lists

Library Lists are saved in your Session Profile, but be aware that saving Library Lists is a **two stage** process:

- First, you have to use the **LIBRARYLISTS CHECKPOINT** command to save the **Active Instance** of the Library Lists data into the in-memory instance of your current Session Profile data. (This will be called the **Saved Instance.**)

- Then you have to harden your changes by using the **PROFILE SAVE name** command to write the profile data out to your profile library on disk. (This will be called the **Stored Instance.**)

Instances of your Library Lists Data

In effect, your Library Lists data exists in up to **three** places:

- There is the **Active Instance.** This is the instance that the **LIBRARYLISTS** command builds and generally manages. It is also the instance that is examined when c/XDC requests Candidate file names and library names from Library Lists Management. There is no specific limit on the aggregate byte size of the Active Instance. (However, see below.)

- There is the **Saved Instance.** This is the instance that gets created by the **LIBRARYLISTS CHECKPOINT** command. It is part of the in-memory instance of your current Session Profile data.

  The Saved Instance **cannot be larger than 8,000 bytes!** If you attempt to use the **LIBRARYLISTS CHECKPOINT** command when the Active Instance is larger than 8,000 bytes, the command will be failed.

  This Saved Instance can also be used as a temporary backup of the Active Instance. If you have issued a bunch of ill considered LIBRARYLISTS commands and, therefore, have managed to generally messed up your lists, you can use the **LIBRARYLISTS CHECKPOINT RESTORE** command to restore the Active Instance from the Saved Instance.

- Finally, there is the **Stored Instance.** This is the hardened copy of the Saved Instance that has been stored to disk as part of your Session Profile's data. It is created by the **PROFILE SAVE name** command. It is the instance that is reloaded whenever you either start a new debugging session or issue a **PROFILE READ name** command.

The important thing to remember here is what was mentioned back at the beginning of this topic... If you want to create a hardened, permanent copy of your Library Lists data, you must issue **two** commands:

- **LIBRARYLISTS CHECKPOINT** copies the **Active Instance** into the Session Profile data area, thus creating the **Saved Instance.**

- **PROFILE SAVE name** saves your Session Profile into your Profile Library, thus creating a **Stored Instance.**

When the Active Instance Gets Too Large

As noted above, the Active Instance of the Library Lists Data can grow to any size; however, both the Saved Instance and the Stored Instance are limited to 8,000 bytes. This of course can be a problem when you wish to create and keep very large Library Lists.

There is, however, a workaround for this. Instead of saving the Library Lists Data into your Session Profile, you can "save" it into a commands script.

That is, you can gather up all of the **LIBRARYLISTS** commands used to create the Library Lists Data, stick a **LIBRARYLISTS RESET** command in front of them, and write them all into a standard Z/XDC commands script.

Then to restore/rebuild the Library Lists Data, just use a **READ scriptname** command to run the script.

Named Session Profiles

Notice that the **PROFILE SAVE** command accepts an optional **name** operand. This allows you to create multiple, named Session Profiles, each having different characteristics. In particular, you can create differing Library Lists and save them into differing Session Profiles. This allows you to, for example, create customized profiles for your various development projects. See **HELP PROFILES NAMEDPROFILES** for more information.

Help DEbugging C Librarylists Searchorder

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

Library Lists are not the only mechanism for finding DWARF files and C source code. A simpler method is to use certain **MAP** command operands:
- **DWARFLIB=** points to a library whose members contain DWARF data.
- **SOURCELIB=** points to a library whose members contain C source code.

DWARF Data Search Order

When searching for DWARF data, c/XDC will search the following places in the following order:

- If you've used the **DWARFLIB=** operand on the MAP command, then that will be the first place c/XDC looks.

- If you have used the **LIBRARYLISTS** command to create lists of DWARF data libraries and folders, then the MAP command will look there next.

- Finally, If the XL C/C++ program's program object contains a hardcoded DWARF library or path/folder name, then c/XDC will try that last.

Notice: The search for DWARF data never checks MAPLIB lists, only Library Lists. MAPLIB lists are used only for finding Assembler Language ADATA, never DWARF data.

C-Source Search Order

Similarly, when searching for C source files, c/XDC will search the following places in the following order:

- If you've used the **SOURCELIB=** operand on the MAP command, then that will be the first place c/XDC looks.

- If you have used the **LIBRARYLISTS** command to create lists of C source libraries and folders, then the MAP command will look there next.

- Finally, If the XL C/C++ or Metal C program's DWARF data contains hardcoded source library or path/folder names, then c/XDC will try those last.

Help DEbugging C Librarylists Management

For a Glossary of Terms, see HELP DEBUGGING C LIBRARYLISTS GLOSSARY.

Library Lists Management

Managing a Library List requires a variety of actions such as:
- Creating/displaying/deleting lists,
- Creating/displaying/deleting list entries,
- Saving/restoring Library Lists Data,
- Running test cases to make sure your lists do what you want them to do.

Accordingly, the **LIBRARYLISTS** command has a number of subcommands as follows:

LIBRARYLISTS ADD {DWARF|CSOURCE} ...
LL
Adds one or more **Result Objects** to an existing or new **Candidates List.**

Result Objects are described in HELP *UP LISTS CANDIDATESLISTS.

LIBRARYLISTS CHECKPOINT
LL CKPT

Creates a **Saved Instance** of the **Active Instance** of the Library Lists Data by copying it into a buffer located within the in-storage instance of Z/XDC's Session Profile Data. See HELP *UP LISTS SAVING for more information.

(Note, creating a **permanently** Stored Instance requires using the **PROFILE SAVE** command.)

LIBRARYLISTS CHECKPOINT RESTORE
LL CKPT

Restores the **Active instance** of the Library Lists Data from a previously saved **Saved Instance.**

LIBRARYLISTS DEFAULT {DWARF|CSOURCE} ...

Names a particular **Candidates List** to be the "default" list. Then when another LIBRARYLISTS command omits any reference to Candidates List, the default list will be the list whose use is implied.

LIBRARYLISTS DELETE {DWARF|CSOURCE} ...
LL

Either deletes **Result Objects** from a Candidates List or deletes **entire Candidates Lists.**

LIBRARYLISTS DELETE {DWARF|CSOURCE} PATTERN=matchpattern
LL

Delete entries from a Family's **Redirect List.**

LIBRARYLISTS REDIRECT {DWARF|CSOURCE} ...

Adds one or more Pattern Match entries to a Family's Redirect List. See HELP *UP LISTS REDIRECTLIST for more information.

LIBRARYLISTS RESET {DWARF|CSOURCE|ALL}

Deletes one or both **Families** of lists. Each Family's **Redirect List** is deleted, along with all **Candidates Lists** owned by the deleted Family.

LIBRARYLISTS SHOW [DWARF|CSOURCE] [ALL] [DETAILS]

Displays one or more Redirect Lists and Candidates Lists either in summary or in detail.

LIBRARYLISTS TEST MATCH ...

Shows the Match Variables that result when a trial Original Name is tested

against a trial Match Pattern.

LIBRARYLISTS TEST {DWARF|CSOURCE} TRY ...
For an existing set of lists, and a given Original Name, this command shows the **Results List** that Library Lists Management would build and return to c/XDC.

Help DEbugging C Librarylists Glossary

Library Lists Management is a rather complex process, and I've had to devise a number of unique terms in order to describe it. The terms are used throughout the **HELP DEBUGGING C LIBRARYLISTS** and the **HELP COMMANDS LIBRARYLISTS** topics, so I guess we need a glossary...

Active Instance [of Library Lists Data]

This is the instance of Library Lists Data that the **LIBRARYLISTS** command builds, maintains and generally manages. It is also the instance that is examined when c/XDC requests candidate file names and library names from Library Lists Management. There is no specific limit on the aggregate byte size of the Active Instance. (But see **Saved Instance, below.)**

For more information see HELP DEBUGGING C LIBRARYLISTS LISTS SAVING.

C Source Code Files
These are one or more datasets containing your C program's source code statements. They are the exact same files used when you ran the XL C/C++ or Metal C compiler to compile your program. c/XDC's map building process needs these files when building Source Image Maps. For more information, see HELP DEBUGGING C LIBRARYLISTS CSOURCE.

c/XDC
This is a License Feature of Z/XDC that adds support to Z/XDC for debugging programs written in XL C/C++ and Metal C.

When licensed, the Feature adds the following capabilities to Z/XDC:

- The **MAP** command is enhanced to read DWARF data and C source code files and build Source Image Maps of your C programs.
- A **LIBRARYLISTS** command lets you build information that may be needed by c/XDC for finding the DWARF data and C source code necessary for building Source Image Maps,
- Several other commands are added for displaying and zapping C variables.

For more information, see HELP DEBUGGING C

Candidates Lists

Candidates Lists are named lists of **Result Objects** that are used to build a **Results List** that will be returned to c/XDC one by one.

Candidates Lists are pointed to either by **Redirect Lists** or by other Candidates Lists.

Any number of Candidates Lists can be created and linked to by a Family's Redirect List and by other Candidates List.

Candidates Lists are built by the **LIBRARYLISTS ADD** command.

For more information, see HELP DEBUGGING C LIBRARYLISTS LISTS CANDIDATESLISTS.

Candidates Names

These are dataset names, file names, library names or folder names that get added to the Results List. They are the names of places that c/XDC is to check when searching for DWARF data or C source data.

Candidate Names are created from **Individual Result Objects** whose Match Variables (if any) have been resolved. For more information, see HELP DEBUGGING C LIBRARYLISTS LISTS CANDIDATESLISTS RESULTOBJECTS.

Collective Result Object

This is the kind of Result Object that represents (either directly or indirectly) a collection of Individual Result Objects, so further resolution will be needed.

There are two kinds of Collective Result Objects:

- The name of another Candidates List.
- A //ddname. The list of datasets allocated to the ddname will be treated as being a Candidates List.

For more information, see HELP DEBUGGING C LIBRARYLISTS LISTS CANDIDATESLISTS RESULTOBJECTS.

CSOURCE

This is the name of a Family of Library Lists pertaining to finding C source code for use by c/XDC.

DWARF

This is the name of a Family of Library Lists pertaining to finding DWARF data for use by c/XDC.

DWARF data file

This is a file that is produced by the C program compilation process. it contains

the information necessary for mapping variable pools and finding C source code files.

c/XDC is capable of reading DWARF data both from classic z/OS sequential datasets and PDS or PDSE libraries, and from HFS files and folders.

For more information, see HELP DEBUGGING C LIBRARYLISTS DWARF.

Families
The Library Lists (managed by LLM) are organized into two Families:

- The **DWARF** Family is a collection of Lists pertaining to finding DWARF data.
- The **CSOURCE** Family is a collection of Lists pertaining to finding C source code files.

Each Family has one Redirect List and its own collection of Candidate Lists.

For more information, see HELP DEBUGGING C LIBRARYLISTS LISTS FAMILIES.

Individual Result Object
This is the name of a dataset, library, file or folder that contains either DWARF data or C source code that c/XDC will need for building Source Image maps.

The name may contain Match Variables that will need to be resolved before it can be considered to be a Candidate Name. But once the variables are resolved, the name can be added to a Results List to be passed back to c/XDC for its examination and consideration.

For more information, see HELP DEBUGGING C LIBRARYLISTS LISTS CANDIDATESLISTS RESULTOBJECTS

Library Lists
These are lists of information that LLM creates and manages for the purpose of providing corrected dataset and file names to c/XDC when it needs to find, open and use DWARF data and C source code.

There are three kinds of lists organized into two **Families.** The Families are named:
- **DWARF**
- **CSOURCE**

The three types of lists are:
- **Redirect List**
- **Candidates Lists**
- **Results List**

All of these terms are defined elsewhere in this Glossary.

Library Lists Data

This is the actual binary data that encodes the Library Lists. It is what gets saved and restored by the **LIBRARYLISTS CHECKPOINT** command.

For more information see HELP DEBUGGING C LIBRARYLISTS LISTS SAVING.

Library Lists Management

This is a collection of services whose underlying goal is to provide to c/XDC the names of datasets or files where it might be able to find DWARF data and C source code that it needs for building Source Image Maps of C programs.

To achieve this goal, you use the LIBRARYLISTS commands to build Match Patterns and file names. Then c/XDC calls LLM services with original file names ("Original Names") and receives back one or more corrected names.

Thorough documentation of Library Lists Management begins at HELP DEBUGGING C LIBRARYLISTS.

LIBRARYLISTS commands

This is a set of c/XDC commands that you can use to build lists of libraries and datasets that c/XDC can search when looking for DWARF data or C source code statements that it needs when building Source Image maps for C programs. See HELP COMMANDS LIBRARYLISTS for more information.

LLM

Short for Library Lists Management.

Match Pattern

This is a string consisting of a mix of constant characters and Wildcard characters (? * and **). Match Patterns are used to accept or reject a Redirect List entry when tested against an Original Name. For more information about wildcards, see HELP DEBUGGING C LIBRARYLISTS LISTS REDIRECTLIST WILDCARDS.

For more information about Match Patterns, see HELP DEBUGGING C LIBRARYLISTS LISTS REDIRECTLIST.

Match Variables

These are variables created by the Pattern Match Process. Their values are set to those portions of the Original Name that were match by wildcard strings from the Match Pattern. One Match Variable is created for each string of consecutive wildcards found in the Match Pattern.

The Match Variables can be used in Individual Result Objects (but not Collective Result Objects). When an Individual Result Objects is about to be added to a Results List, Any variables present are first resolved, thus transforming the Individual Result Objects into a customized Candidate Name.

For more information see HELP DEBUGGING C LIBRARYLISTS LISTS REDIRECTLIST MATCHVARIABLES.

Original Name
This is the name of a DWARF data file or C source code file as it existed at the time that a C program was last compiled and bound. This name is what c/XDC passes to LLM when it needs to know if the name needs to be corrected.

For information about original DWARF file names (and why they might need to be corrected), see HELP DEBUGGING C LIBRARYLISTS DWARF.

For information about original C source file names see HELP DEBUGGING C LIBRARYLISTS CSOURCE.

Pattern Match Process
This is the process of testing Original Names (from c/XDC) against the Match Patterns of Redirect List Entries and building a Results List based on those patterns against which the Original Name matched. For more information see HELP DEBUGGING C LIBRARYLISTS LISTS REDIRECTLIST.

Redirection
In order to build a C program's Source Image Map, c/XDC needs to find, open and read a DWARF data file as well as one or more C source code files. At the time a program was compiled and bound, the DWARF data was created in a dataset or file whose name wound up being hardcoded into metadata in the program object.

If for some reason, no name was hardcoded, or if the DWARF data or the C source code was subsequently moved to another place, then the hardcoded name is either missing or no longer correct. In these cases, c/XDC's knowledge (or lack thereof) of the file name has to be **corrected,** and its process of opening and using the DWARF data (or C source code statements) has to be **redirected** to the correct dataset.

For more information, see HELP DEBUGGING C LIBRARYLISTS DWARF.

Redirect List
This is a single list that consists of one or more entries that connect Match Patterns to named Candidates Lists of files, folders, datasets, libraries and //ddnames that c/XDC is to search for the needed DWARF data or C source file.

There is one Redirect List per Family (of lists).

Redirect Lists are built by the **LIBRARYLISTS REDIRECT** command.

Result Object
All entries in a Candidates List are **Result Objects.** All entries in a Redirect List are Match Patterns Linked to Result Objects. Result Objects ultimately are used in the construction of a **Results List.**

A Result Object will be either a Candidate Name or an object that eventually will be resolved into a **collection** of Candidate Names. There are two kinds of **Collective** Result Objects:

- The name of another **Candidates List,**
- A **//ddname.** (The list of datasets allocated to the ddname will be treated as being a Candidates List.)

For more information, see HELP DEBUGGING C LIBRARYLISTS LISTS CANDIDATESLISTS RESULTOBJECTS.

Results List
This is a temporary list that Library Lists Management builds when resolving an Original Name that it has received from c/XDC. The list entries consist of resolved Candidate Names that LLM passes one by one back to c/XDC for its consideration as a substitute for the Original Name of the DWARF file or C source code file that it wants to open.

For more information, see HELP DEBUGGING C LIBRARYLISTS LISTS RESULTSLIST.

Saved Instance [of Library Lists Data]

This is the instance of Library Lists Data that gets created by the **LIBRARYLISTS CHECKPOINT** command. It is part of the in-memory instance of your current **Session Profile** data. Unlike the Active Instance, the saved instance has a size limit: 8,000 bytes. If the Active Instance grows too large, it cannot be saved.

Important! This instance has not yet been hardened to disk. For more information see HELP DEBUGGING C LIBRARYLISTS LISTS SAVING.

Stored Instance [of Library Lists Data]

This is the hardened copy of the Saved Instance that has been stored to disk as part of your Session Profile's data. It is created by the **PROFILE SAVE name** command. It is the instance that is reloaded whenever you either start a new debugging session or issue a **PROFILE READ name** command.

For more information see HELP DEBUGGING C LIBRARYLISTS LISTS SAVING.

Wildcards
Wildcards can be given within Match Patterns that are used to accept or reject a Redirect List entry when tested against an Original Name. There are three kinds of wildcards that LLM understands:

- **?** matches exactly one character,
- * matches 0 or more characters up to the next period, open parenthesis, close parenthesis or forward slash [**.) (** /].
- ** matches 0 or more characters up to the end of the Original Name from c/XDC.

For more information see HELP DEBUGGING C LIBRARYLISTS LISTS REDIRECTLIST WILDCARDS

Help DEbugging C Debugging

When a c/XDC debugging session starts, if both **AUTOMAPing** and **autostepping** are turned on, program execution will automatically be advanced (through all of the LE prolog code) to the start of your actual **C** program code. See HELP COMMANDS STEP for more information.

On the other hand, if either **AUTOMAPing** or **autostepping** is turned off, then program execution will be intercepted exactly at your load module's entry point.

Similarly, when the debugging session starts, if **AUTOMAPing** is turned on and DWARF data is available, then your program's module and Source Image maps will automagically be loaded. See HELP COMMANDS AUTOMAP for more information.

On the other hand, if **AUTOMAPing** is turned off, then you will have to issue the **MAP** command manually, and stepping (auto or otherwise) cannot occur until a map has been loaded.

When you issue the **MAP** command, you will see a report that looks something like this:

```
      The following maps have been built:
      MAP TYPE          LOCATION    NAME
   _ PROGRAM OBJECT    19B76000    modulename (c)
   _ C SOURCE          19B76000    controlsectionname_C_CODE (c)

      (c) - DENOTES MAPS WHOSE LABELS ARE CASE-SENSITIVE.
```

If you type an **F** ("Format") shortcut into the Shortcut Entry Field of the line captioned **C SOURCE,** you will then see the source code for your C program.

You can now type a **Q** shortcut command into the Shortcut Entry Field for pretty much any displayed source statement. This will make your C program the default that z/XDC will use when resolving address expressions.

Now, you can issue **FORMAT** commands that reference the C statements by statement numbers. Examples:

- F .#90
- F .B32758

These bring into view statement numbers #90 and B32758, respectively.

Note, c/XDC displays statement numbers are led either by **#** or by an alpha character. These leader characters are used by c/XDC to disambiguate sequence numbers across source files. So you need to use them on **FORMAT** (and other) commands when referencing language statements by statement number.

If you would like to get an overview of your program object and the functions and subroutines that it contains, you can do the following:

- Use an **L** shortcut command (LIST LKEDMAP +0) at the left to produce a display of the program object's binder map.

- Find your function's name in the map and place an **F** shortcut command (FORMAT +0) at the left to see the function's source code.

- If the function has not yet been mapped, you will see only disassembled machine code, in which case...
  - You can use an **M** shortcut command (MAP +0) to load the source image map.
  - Then issue a **FORMAT +0** command (on the command line) to see the source code.

If you wish to scroll down through your source listing, just press **F8** (a DOWN command). To scroll up, use **F7.**

You can set an execution trap in your C program by typing a **T** into the Shortcut Entry Field for whichever source statement you want execution to stop at.

Now type **GO** (or **GOT** if you're debugging with your C program running in TSO). Your program's execution will begin, and it will run until it reaches your trap.

LIST VARIABLES vref vref ...
LIST VARIABLES ALL
At any point, you can use this command to display the current values of one or more currently active XL C/C++ and Metal C variables, structures and arrays. The variables are displayed and formatted according to their types. For more information, see HELP DEBUGGING C VARIABLES.

You can also display variables found in older stack frames for the routines that called the current instance of the current routine.

The resulting displays will have Shortcut Entry Fields in which you can issue **D** ("Display") or **F** ("Format") shortcuts to display the actual storage in which the variables reside. If you want to **zap** a variable's value, you can do so using **Z** ("Zap") shortcuts.

STEP INTO
This command is used to step through execution one XL C/C++ or Metal C statement at a time. When the current statement calls a subroutine or function, and **if that subroutine is mappable,** then this command (AUTOMAP permitting) maps the subroutine and then causes execution to step into that subroutine or function. (If that subroutine or function is not mappable, then it is stepped over, not into.)

STEP OVER
This command also is used to step through execution one XL C/C++ or Metal C statement at a time. But when the current statement calls a subroutine or function, this command allows that subroutine/function to run, and execution is stopped upon return on the next statement following the subroutine/function call.

STEP OUT
When execution is within a subroutine or function, this command can be used to allow the rest of the subroutine/function to execute to completion. Execution is stopped when the subroutine or function returns to its caller. (When issued from **main(),** STEP OUT is disallowed.)

STEP
When the **STEP** command is given without operands, it will default either to STEP **INTO** or to STEP **OVER,** according to the current **SET STEP** setting.

Help DEbugging C Variables

The command used to display XL C/C++ and Metal C variables is **LIST VARIABLES vref vref [...]** where one or more **vrefs** reference the variables to be displayed. The references can be any of the following.

- They can be names that reference any kind of C variable.
- For structures, the references can be to the entire structure or to individual members (which themselves, of course, can be structures or arrays).
- For arrays:
  - The references can be to individual members or selections of members.
  - The selections can be lists or ranges (or both).
  - The ranges can be vectors or rectangles or cubes or generally any n-dimensional block of contiguous members.
  - The members to be displayed can be specified either via integer constants or numeric variables. For numeric variables:
    - Non-integer values are floor'd down to the next lower integer.
    - Character strings consisting of a numeric value are permitted.
    - Pointer variables can be used as well.
    - Expressions are **not** supported.

If you simply want to see a display of all variables, **LIST VARIABLES ALL** will to the job for you.

Rules for Arrays - Displaying an Entire Array

To display an entire array, just provide the array's name. Examples:
- **L V MyArray**
- **L V MyStructure.MyArrayWITHINMyStructure**

To control the accidental display of huge arrays, the LIST VARIABLES command's display can be limited by two operands of the **SET VDISPLAY** command:

- SET VDISPLAY **VLINES=nnn** limits the number of display lines that an array is allowed to take. The display of the array terminates after said number of lines has been produced.
- SET VDISPLAY **LINES=nnn** limits the total number of lines that can be produced by a LIST VARIABLES command.

The current values of these settings can be displayed by the **LIST VDISPLAY** command.

The values are saved in your session profile, so they can be loaded, saved and displayed by the Profile Menuing System. See HELP PROFILES for details.

Rules for Arrays - Displaying Character Strings

If the array is a 1-dimensional character string, (or a 2-dimensional collection of 1-dimensional character strings, or etc.), then it will be displayed as a string (not as discrete members). Note, if the array contains a null delimited string, then once the display is presented...

- You can use the \ shortcut command to show only the string up to the null delimiter.
- You can use the / shortcut command to show the entire array of characters. (The display does not stop at the null.)

When a character string is too long to be displayed within a single line, you can use the **L** shortcut command to toggle back and forth between single line displays and multiline displays.

Character strings can be displayed either in ASCII or EBCDIC. Use the * and | shortcut commands to set one or the other.

Rules for Arrays - Displaying Selections of Individual Members

To show just a single member, specify the indices for that member. Example: For a 3 dimensional array named MYCUBE, use (for instance) **L V MYCUBE[12][7][3]**

You can use numeric variables instead of constants. Example: **L V MYCUBE[H][W][3]** - Notes regarding variables:

- Non-integer values are floor'd down to the next lower integer.
- Character strings consisting of a numeric value are permitted.
- Pointer variables can be used as well.

To display a selection of members, use a comma delimited list of numbers (or variables) for one or more of the indices. Example: **L V MYCUBE[1,7,4][3,2][3]** - This displays six members from MYCUBE.

To display a range of members, use a [number:number] syntax. Example: **L V MYCUBE[3:6][2:4][5]** - This displays a 4x3 rectangular slice of MYCUBE. Note, specifying backwards ranges (such as [6:3]) is permitted.

All of the above can be combined. Example: **L V MYCUBE[3:N,9][2,W][5,10:8]**

Managing the Display of Structures

To display a member within a structure, specify its name qualified by the structure's name. Examples:

- **L V MYSTRUCTURE.THISMEMBER**
- **L V MYSTRUCTURE.INNERARRAY[3][4].INNERSTRUCTURE.MEMBER**

To display an entire structure, start by referencing the structure's name. Examples:

- **L V MYSTRUCTURE**
- **L V MYSTRUCTURE.INNERARRAY[3][4].INNERSTRUCTURE**

A single line will be displayed with one of the following captions:

*** Structure not Explored *** or
*** Structure is Hidden ***

- Then use the **L** shortcut command to drill down one level into the structure. (You also use the **L** shortcut to climb back out of a drill-down.)

Help DEbugging C CXDCHook

- Hooks are used for inserting a debugging session into code where one does not already exist. (This is as opposed to breakpoints which can be used only after a debugging session already exists.) See HELP HOOKS for more information.

- If you wish to insert a dynamic hook into your C program, just use the **HOOK** command just as you would for an Assembler program.

- On the other hand, if you wish to compile a **static** hook into your C program (like what the #XDCHOOK macro does for Assembler programs), your code needs to call a function that we provide named **cxdchook.** (See HELP HOOKS STATIC for more information about static hooks.)

cxdchook has two parts:

- **CXDCHOOK** is a C language header file that you need to include into your C program. It can be found in the hlq.XDCV31.XDCSAMP sample library. (Ask your Systems Programmer for the library's actual name at your Data Center.)

- **AXDCHOOK** is a load module that is located in the hlq.XDCV31.XDCLINK load library. (Again, ask your Systems Programmer for the library's actual name at your Data Center.) This is the actual code called as the cxdchook function.

  The source code for AXDCHOOK can be found in hlq.XDCV31.XDCASM(SRCAHOOK).

- Basically, the CXDCHOOK function invokes the **#XDCHOOK** macro which creates the debugging environment and then passes control to Z/XDC. (See HELP HOOKS STATIC #XDCHOOK for more information.)

We also provide a sample C program named **XDCCSAMP** that illustrates how to use CXDCHOOK. Its source code is in hlq.XDCV31.XDCSAMP(SRCCSAMP). Its load module is in hlq.XDCV31.XDCLINKE(XDCCSAMP).

CXDCHOOK Compile Time Requirements

In order to call CXDCHOOK, you need to make some changes to your C Program:

- You need to add **LSEARCH(//'hlq.XDCV31.XDCSAMP')** to your compile time options file. See hlq.XDCV31.XDCJCL(ASMCSAMP) for sample JCL.

- You need to include The **cxdchook** header file into your program. The following statement will do the trick: **#include "cxdchook.h"**

- Then at the point in your code where you want to start a debugging session, you need to call **cxdchook.** Use this statement: **cxdchook("XDC");** When execution

passes to the function, an XDC debugging session will be started.

- If you are using a **clone** of XDC named (for example) **V31,** then the statement you need to use is: **cxdchook("V31");**

CXDCHOOK Bind Time Requirements

The CXDCHOOK function's code is contained in a load module named **AXDCHOOK** (located in the load library named hlq.XDCV31.XDCLINK). In order for the Binder to find the AXDCHOOK load module, you need to:

- Add hlq.XDCV31.XDCLINK to your **//SYSLIB** concatenation.
- Add an INCLUDE statement to the Binder's **//SYSLIN** input file. Example: **INCLUDE SYSLIB(AXDCHOOK)**

Providing a Clone Name to CXDCHOOK

Normally, Z/XDC's name is "XDC", **but** it is possible to renamed XDC to any other 3-character name. Such a name is called a **clone name,** and the purpose of clone names is to allow multiple releases of Z/XDC (as well as multiple instances of the same release of Z/XDC) to coexist gracefully on the same system. So the effects of switching Z/XDC's name to a clone name are extensive! For details, see HELP XDCCLONES.

Here at ColeSoft, we use clone names all the time in our development process, but this is not something that a customer normally would need... **except** when we publish **beta levels** of a new release of our products. In this case, in order to allow a beta to coexist with an older production release, we give the beta a suitable 3-character name. So for example, the beta release of Z/XDC v3.1 might be named **V31.**

Consequently, the **cxdchook** function needs to be able to react to clone names. Accordingly, you may have noticed above that cxdchook accepts a string argument. That string may be either null, or it may contain a 3-character clone name. Example: **cxdchook("V31");**

The **XDCCSAMP** sample calls another function named **cxdcis()** to determine the desired XDC clone name, and then it passes that name to **cxdchook** for creating the debugging session under the right Z/XDC clone. For more information, see HELP DEBUGGING C CXDCIS.

Help DEbugging C CXDCIs

Normally, Z/XDC's name is "XDC", **but** it is possible to renamed XDC to any other 3-character name. Such a name is called a **clone name,** and the purpose of clone names is to allow multiple releases of Z/XDC (as well as multiple instances of the same release of Z/XDC) to coexist gracefully on the same system. So the effects of switching Z/XDC's name to a clone name are extensive! For details, see HELP XDCCLONES.

Here at ColeSoft, we use clone names all the time in our development process, but

this is not something that a customer normally would need... **except** when we publish **beta levels** of a new release of our products. In this case, in order to allow a beta to coexist with an older production release, we give the beta a suitable 3-character name. So for example, the beta release of Z/XDC v3.1 might be named **V31.**

To deal with this, we provide a function named **cxdcis()** that can be used to determine what clone name should be used.

Syntax:

```
string = cxdcis()
```

cxdcis() does not accept any input parameters. The value returned by the function is always an 8-character string whose value is the 3-character name to be used for Z/XDC, padded on the right to 8 characters by blanks. Usually, this value will be **XDC,** but it can be any other 3-character string value that is legal as a name according to JCL PDS member name rules.

cxdcis() works by searching the current jobstep for a ddname of the form **//XDCISxxx,** and then it returns the value of **xxx** as the clone name.

Example: If **cxdcis()** finds the ddname **//XDCISV31,** then it will return to its caller the value **V31.**

The easiest way to create an //XDCISxxx allocation in the batch is to provide a **DD DUMMY** JCL card. Example: **//XDCISV31 DD DUMMY**

In TSO, you would issue an **ALLOCATE** command more or less as follows: **ALLOC FILE(XDCISV31) DUMMY REUSE**

It is important to realize that neither **cxdcis()** nor Z/XDC ever OPEN's an //XDCISxxx allocation. It only checks for the allocation's presence or absence.

The following special and error conditions can arise...

- An //XDCISxxx allocation is not present: (This is a special condition, **not** an error condition.) **cxdcis()** returns the default value: **XDC.**

- The **xxx** portion of //XDCISxxx is less than three characters long: (This is an error condition.) **cxdcis()** just returns the default value: **XDC.**

- The **xxx** portion of //XDCISxxx starts with a digit: (This is an error condition.) **cxdcis()** just returns the default value: **XDC.**

The **cxdcis()** function is distributed in two parts:

- One part is a **header** file named **cxdcis.** It can be found in hlq.XDCV31.XDCSAMP(CXDCIS). It defines the attributes of the **cxdcis()** function. It must be included into your C program via an **include** statement. Example: **#include "cxdcis"**

  You also will need to add **LSEARCH(//'hlq.XDCV31.XDCSAMP')** to your compile time

options file. See hlq.XDCV31.XDCJCL(ASMCSAMP) for sample JCL.

- The second part is a load module named **AXDCIS** (not "CXDCIS" - Note, the cxdcis header file notifies the C compiler of this.) This load module is located in hlq.XDCV31.XDCLINK(AXDCIS). If you want to find out more about **AXDCIS,** see HELP XDCCLONES AXDCIS.

  At Bind time, you will need to add the following:
  - You will need to add hlq.XDCV31.XDCLINK to your //SYSLIB concatenation.
  - You may need to add INCLUDE SYSLIB(AXDCIS) to the //SYSLIN input file.

  See hlq.XDCV31.XDCJCL(LCSAMP) for sample JCL.

- In addition, the source code (written in Assembler) for AXDCIS can be found in hlq.XDCV31.XDCASM(SRCAXDCI).

You can find an example of using the cxdcis() function in hlq.XDCV31.XDCSAMP(SRCCSAMP).

Note, the above given library names are our factory default names. You will have to ask your Systems Programmer what their actual names are at your Data Center.

Help DEbugging C Externalissues

As of January 2016, the following are known problems within c/XDC that are due to issues within the external interfaces that we employ:
- The XL C/C++ compiler
- The Common Debug Architecture (CDA) API
- The pre-linker

We are working with IBM to rectify these issues as quickly as possible.

Problem

The Metal C compiler does not generate correct DWARF for pointer variables when:
- The variable is defined with the **far** attribute (such as when the variable resides within a dataspace),
- And the pointer variable is a member of a larger defining structure.

In this case, c/XDC does not the display the correct storage for the pointer target.

Impact: No current solution. Many Metal C structure variables that involve pointers will be assigned incorrect addresses and/or ALETs. No c/XDC workaround is available.

We have opened this issue with IBM The PMR is =======.

Problem

The information contained within DWARF data regarding statement labels (example **mylabel:)** does not contain any information about the location of the label. DWARF basically says that **mylabel** is defined somewhere within the source, but not where. Therefore, there is no support yet in c/XDC for reference-by-label.

Impact: Statement labels are either unavailable or non-useful as a name c/XDC can recognize to represent a location within the program. No c/XDC programming workaround is available.

We have opened this issue with IBM The PMR is =======.

Problem

Sometimes regarding lexical blocks (FOR groups, IF-THEN-ELSE clauses, and the like) there is a mismatch between what the DWARF data says are the block's boundaries and what the boundaries actually are. Specifically, sometimes the DWARF data says that a block ends before it actually does.

Impact: When execution is located on a machine instruction that is within the block but the DWARF thinks otherwise, variables whose scope is limited to the block can no longer be displayed even though they still exist.

We have opened this issue with IBM The PMR is =======.

Help DEbugging CIcs

Since Z/XDC is a generalized Assembler debugging tool, it can be used to debug transactions and exit routines running within CICS. Z/XDC, however, has only limited support for and knowledge of CICS. Most of Z/XDC's dependencies and interfaces are based upon the z/OS operating system in general, not CICS.

This design gives rise both to Z/XDC's power and limitations in a CICS environment. Because Z/XDC does not use (and in fact is not even aware of) CICS service routines, it can be used to debug just about **any** code running anywhere in CICS: user transactions, installation exit routines, and the CICS service routines themselves. The major restrictions are the following:

- When Z/XDC is in control, CICS is not; therefore, the rest of the CICS address space is frozen; therefore, all users attached to that copy of CICS suffer locked keyboards at their terminals. Consequently, Z/XDC can be used only with private copies of CICS where the Z/XDC user is the only user connected to that CICS.

Z/XDC does, however, have a sufficient understanding of CICS to be aware of where all of CICS-loaded load modules are. This makes it possible to load maps for any module, to use CICS-loaded module names in address expressions, and to see module information in storage displays.

Currently, Z/XDC's ability to find CICS-loaded load modules is supported only in these CICS releases: 4.2 through 5.6.

Two Terminals Will be Needed

Since Z/XDC does not use CICS services, it cannot communicate with its user via the CICS terminal. Instead, a second logical or physical terminal must be used. In most cases that will be a TSO terminal connected to Z/XDC via cdf/XDC. One benefit of this is that Z/XDC's displays and terminal management techniques can never interfere with CICS's.

Z/XDC's CICS Transaction

In order to use Z/XDC, CICS's ABEND error recovery routines must be disabled. These routines consist of an ESPIE exit for program checks and an ESTAE exit for other ABENDs. ColeSoft provides a pre-written transaction program that can be used for this purpose. Once this transaction is executed, all system ABENDs, and all nonmathematical program checks will be passed to Z/XDC for processing instead of to CICS. This transaction can be executed at any time, including manually from the CICS terminal, automatically during CICS's stage-3 initialization (as a PLT-PI routine), or via any other mechanism permitted by CICS.

To Summarize ...

In order to use Z/XDC to debug in CICS, you need the following:
- A private copy of CICS with your CICS terminal defined to it.
- An Z/XDC setup transaction defined to that CICS.
- Access to two logical or physical terminals, one for CICS, and one Z/XDC. The Z/XDC terminal can be either a TSO terminal or even just an otherwise idle VTAM terminal.
- cdf/XDC installed and running on your z/OS system.
- Minor modifications to the JCL in your CICS startup proc to allow Z/XDC to interface to its Cross Domain Facility so that Z/XDC can communicate with its terminal.

Step-by-Step Setup

The following is a detailed step by step procedure required for using Z/XDC in CICS.

- **cdf/XDC:** Ensure that cdf/XDC is installed and running. This is documented in Z/XDC's "Installation Guide".

- **JCL:** Create your private copy of CICS, and modify its startup proc as follows to allow Z/XDC to connect to cdf/XDC:

```
//name   EXEC PGM=DFHSIP, ...
//STEPLIB  DD DISP=SHR,DSN=hlq.XDCV31.XDCLINK
//         DD DISP=SHR,DSN=hlq.XDCV31.XDCPLPA
//         DD <additional libraries, as may be needed by CICS>
//xxxPROF  DD DISP=SHR,DSN=profile.lib
//XDCISxxx DD DUMMY
//  <other DD cards required by CICS>
```

  - **xxx** must match Z/XDC's current name. Usually, that would be **XDC,** but it could be any legal 3-character name, like DBC, V31, your initials, whatever. For more information, see **HELP XDCCLONES.**

- When XDC is named something other than "XDC", the **//XDCISxxx** DD card is needed to notify XDC what its name is. (Otherwise, this DD card can be omitted. In other words, providing an **//XDCISXDC** DD card, while permitted, is never necessary.)

- The **xxxPROF** DD card should point to a library that contains or that will contain user profile information. Its DCB attributes must be **RECFM=FB, LRECL=80** and **BLKSIZE=n*80.** This DD card is optional. If omitted, then Z/XDC will use one of the **factory default** profiles.

- Normally, Z/XDC's load modules are installed into the System's linklist and LPA libraries. However, if they are not, then you will need to add Z/XDC's distribution libraries to CICS's **//JOBLIB** or **//STEPLIB** concatenation. The factory default names for those libraries are:
  hlq.XDCV31.XDCLINK
  hlq.XDCV31.XDCPLPA
  However, your Data Center's Systems Programmer may have chosen different names. So you may have to ask him for the actual dsnames.

- **The XDC Transaction:** The Z/XDC setup transaction is in a load module named **xxxCICSX** (where **"xxx"** matches Z/XDC's current name, usually "XDC"). You must add this module name to CICS's program table and assign to it a suitable transaction name. (How about "XDC"?) Make sure that the transaction is assigned to CICS storage, not to user storage.

- **Protection:** Make sure that CICS's S.I.T. has **RENTPGM=NOPROTECT** specified.

- **Startup:** Start your CICS and signon to it.

- **Starting up Z/XDC:** Run the XDC transaction to disable CICS's error recovery in favor of Z/XDC and to pass control to Z/XDC (or to a Z/XDC clone, according to a **//XDCISxxx** DD card). This will have the following effects:

  - Your CICS terminal will go dark, and your keyboard will lock up.

  - Z/XDC will connect to cdf/XDC and wait there for you to signon to the debugging session. During this wait the following WTOs/WTOR messages will appear at the System Operator consoles:

```
    +DBC640Q XDC v3.1: (jobname) AWAITING USER SIGNON.
     DBC640Q    - Reply C to cancel the wait and percolate the ABEND.
     DBC640Q    - Reply GO to cause the user's program to reattempt
                  execution.
     DBC640Q    - Reply WTOR to issue z/XDC commands via WTO/WTOR.
     DBC640Q    - Reply RESHOW to redisplay these messages.
   @nn DBC640Q XDC v3.1: (jobname) Reply C, GO, WTOR, or RESHOW
```

- **Connecting to Z/XDC:** At this point you must switch to another terminal in order to connect to the debugging session via cdf/XDC. There are two ways to connect to cdf/XDC:
  - Logon to TSO.
  - Go into ISPF.
  - Find Z/XDC's Startup Panel in ISPF. (Ask your Systems Programmer where in ISPF he installed the Z/XDC Startup Panel.)
  - Ensure that the panel's **XDC's name ===>** field matches Z/XDC's current clone

name.
- Select Option **3.**
- Press ENTER.

OR
- Get to idle VTAM terminal.
- Log directly on to cdf/XDC by typing one or the other of the following:
  LOGON APPLID=xxx
  LOGON APPLID(xxx)
  (where **xxx** matches Z/XDC's current name, usually "XDC").
- cdf/XDC will then display a logon screen where you have to type in your TSO userid and password.

Once you have connected to cdf/XDC, you will see a **Job Selection Menu** which should be displaying your CICS's jobname. (If not, then you have security access rule definition problems. See **HELP SECURITY CDF** for more information.) Type an **S** next to the jobname. You will now be connected to the debugging session for your CICS job.

- **Setting and Reaching Traps:** Use Z/XDC's **AT** and **TRAP** commands to set persistent (AT) or transient (TRAP) breakpoints at the locations in your program that you want to debug. Example: **AT .TOPLOOP.**

Use Z/XDC's **GO** command to return control to CICS. This will have the following effects:
- Your Z/XDC terminal's keyboard will lock.
- The Z/XDC setup transaction in CICS will resume execution and immediately terminate back to CICS.
- The CICS terminal's keyboard will unlock and be ready to accept another transaction name.

At the CICS terminal, type the transaction name that will cause your program to gain control. Do whatever is necessary to cause the breakpointed program code to be executed. When this occurs, Z/XDC will again gain control, and the CICS keyboard will again lock up, and the Z/XDC keyboard will unlock. You are now ready to debug your code.

- **Conducting Your Debugging Session:** You can use the **TRACE, TRACE BY** and **TRACE BNC** commands to cause execution to step through your program one or a few instructions at a time.

You can use the **DMAP** and **USING** commands to read control block maps and assign them to locations in storage.

You can use the **FORMAT, DISPLAY, SHOW** and **WHERE** commands to display storage in various ways.

- You can use the **LIST REGS** and **LIST AREGS** commands to display the general registers and access registers, respectively.

- You can use Z/XDC's **Z** ("Zap") shortcut to change displayed data and machine instructions.

See **HELP COMMANDS** for a complete list of commands available in Z/XDC.

Subtopics Index

For more information, select the following topic. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

XDCCICSX - An Z/XDC setup transaction for CICS.

Related information can be found by typing:
    HELP MAPS CICSMAPS

Help DEbugging CIcs Xdccicsx

XDCCICSX is a load module that contains a CICS transaction that sets up CICS for Z/XDC debugging. It does this by overriding CICS's normal program check intercept routine (ESPIE) and ABEND error recovery routine (ESTAE) so that control is passed to Z/XDC instead of to CICS when these events occur.

To install the transaction, you have to assign to the XDCCICSX load module a transaction name (such as "XDC") by adding an entry into CICS's program table. Make sure that you define the transaction to execute in "CICS key", not in "user key".

To activate Z/XDC, you have to invoke the "XDC transaction". This can be done manually from the CICS terminal, automatically during CICS's stage-3 initialization (as a PLT-PI routine), or via any other mechanism permitted by CICS.

If you are using a Z/XDC whose name has been changed from XDC to some other 3-character name (a "clone name"), then you will have to add the following DD card to your CICS's JCL:

```
   //XDCISxxx DD DUMMY
```

where "xxx" is XDC's clone name. For more information, see HELP XDCCLONES.

Running the XDC Transaction

When you run the transaction for the first time, it will do the following:

- It modifies CICS's ESPIE to intercept only the mathematical program checks (interrupt codes 0008 through 000F) and not to intercept program interrupt codes 0001 through 0007. This allows these program interrupts to be forwarded to ESTAE processing as ABEND codes s0C1 through s0C7.

- It issues an ESTAE macro making Z/XDC the newest ABEND error recovery routine for CICS. This effectively nullifies CICS's own ESTAE.

- It executes an assembled breakpoint (a #DIE macro) to cause control to pass to Z/XDC so that initial maps can be loaded and initial breakpoints can be set.

When you run the transaction for a second or subsequent time, it only executes the #DIE macro. It recognizes that CICS's ESPIE and ESTAE have already been suppressed and so does not do it again.

When the transaction passes control to Z/XDC (via the #DIE macro), all of CICS is

interrupted and all terminals connected to that copy of CICS will lock up. Z/XDC will issue its displays through a non-CICS interface, such as via cdf/XDC to a TSO terminal. CICS will remain locked up until you issue Z/XDC's "GO" command to allow CICS execution to resume.

Help DEbugging Dumps

What is dump/XDC? And what is its Value in an IPCS world?

Briefly, **dump/XDC** is a Licensed Feature that allows you to troubleshoot IPCS dumps with the same powerful Z/XDC command set that you use to debug running programs in live storage.

dump/XDC does not replicate IPCS's commands and capabilities, but likewise, IPCS does not replicates Z/XDC's. Instead, dump/XDC augments IPCS's power with its own considerable abilities.

For comprehensive doc about dump/XDC, take a look at **HELP DUMPS.**

Help DEbugging EStaes

Z/XDC is a powerful debugger. It gains its power from the fact that it normally runs as an **ABEND Recovery** routine (ESTAE, ESTAI and the like).

Z/XDC also can be setup to run as a **Trap Handler** (for **TRAP2** machine instructions). But even when it is a Trap Handler, it still must **also** be an ABEND Recovery Routine. For more information, see HELP BREAKPOINTS TYPES.

Dealing with Interference from Other ABEND Recovery Routines

But Z/XDC, as powerful as it is, can be disrupted when the program that is being debugged sets up its own ABEND Recovery routines (ESTAE, ESTAI, ARR, FRR, etc.) and/or Program Check Intercept routines (SPIE and ESPIE routines). The following kinds of problems arise:

- ABEND Recovery routines, when **newer** than the ESTAE or ESTAI routine that is Z/XDC, will intercept ABENDs or breakpoints ahead of Z/XDC. From a debugging point of view, such interference will almost always have a catastrophic result! Leading to inexplicable failures and other hilarities.

- Program Check Intercept routines **(ESPIEs),** if they exist at all, will always receive control ahead of any ABEND Recovery routine, including Z/XDC when it runs as an ABEND Recovery routine. Again, mishandling and other hilarities will occur. (When Z/XDC runs as a Trap Handler, this problem is less likely to arise.)

Therefore, when these conditions exist, Z/XDC's usefulness becomes severely

compromised.

If a program already has ESTAEs, ESTAIs, ARRs, ESPIEs and/or FRRs of its own, then in order to use Z/XDC, steps will have to be taken to either get in front of or not use all user program recovery and intercept routines. Setting Z/XDC up to run as a **Trap Handler** can mitigate the problems, but since Z/XDC also has to be set up as an ABEND Recovery routine anyway, problems can still arise.

So generally, the best solution is to make sure that an instance of Z/XDC is set up as an ESTAEX routine **in front of** all of your program's own ABEND Recovery routines. Usually, **the easiest way** to do this is to place **hooks** into the code to be debugged.

A **hook** is a special bit of code that is temporarily overlaid onto user code at the point at which debugging is to be started or resumed. When user execution reaches this hook, a suitable Z/XDC environment is created and debugging can be commenced (or resumed).

Hooks can be created at assembly time by the **#XDCHOOK** macro or at run time via Z/XDC's **HOOK** and **HDEFERRED** commands. For more information, see **HELP HOOKS.**

Integrating Z/XDC into Existing ABEND Recovery Routines

As just mentioned above, **hooks** are very easy to use. But... if you are involved in the long term development of a large complex program, or if you find that you need to set up a debugging structure for use by other less experienced programmers, that you may find it advantageous to make changes to your source code in order to install permanent Z/XDC support into your program. If this is your choice, then consider the following:

- If your program already has ESTAEs, ESPIEs, ARRs, FRRs (etc.) of its own, then either the recovery routines will have to be removed, or they will have to be superseded. Or they will have to be modified to call Z/XDC under appropriate conditions.

  Of these several choices, modification of your existing recovery routine may be the most flexible in the long run.

- If YOUR program already has SPIEs or ESPIEs, then they too will have to be modified. Either the SPIE/ESPIEs will have to be removed, or they will have to be modified to intercept only the mathematical program checks. (Superseding SPIE/ESPIEs is not possible in the same sense that ESTAE/ESTAIs can be superseded.)

Subtopics Index

The following topics explain various methods for coping with other ABEND Recovery routines. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

REMOVING - Removing user program ESTAE/ESTAIs that interfere with Z/XDC.
SUPERSEDING - Superseding user program ESTAE/ESTAIs that interfere with Z/XDC.
MODIFYING - Modifying user program ESTAE/ESTAIs that interfere with Z/XDC.

ESPIES - Dealing with user program SPIE/ESPIEs that interfere with Z/XDC.
HOOKS - Using hooks to overcome interfering ESTAE/ESTAI/ARRs and SPIE/ESPIEs.
DEBUGGING - Using Z/XDC to debug your ESTAE, ESTAI, SPIE, and ESPIE routines.

Help DEbugging EStaes Removing

ESTAE/ESTAI removal can be accomplished either by source code changes at program assembly time or by a file of Z/XDC **ZAP** commands at execution time. Of these choices, creating a file of Z/XDC **ZAP** commands is probably more flexible because then you don't have to reassemble your program just to turn your ESTAEs on and off.

For information about Z/XDC command scripts, see **HELP COMMANDS READ.**

To build a Z/XDC command script to disable your ESTAE/ESTAIs, proceed as follows:

- If you have assembly listings for the program you are working with, and if you are reasonably familiar with its logic, then you can find and write down the locations of your ESTAE SVCs and/or your ESTAEX PCs. HINT: You should record these locations relative to the start of your csect or, better yet, relative to a nearby label. Then your **ZAP** commands can use those labels in their target address expressions. Then you will not have to change your **ZAP** commands every time you reassemble your program. (See **HELP MAPS** for more information about using labels from your program.)

- Using an editor, create a PDS member (or sequential file) that contains the Z/XDC **ZAP** commands that change the ESTAE SVCs (or ESTAEX PCs) into no-ops. HINT: Sometimes an ESTAE SVC will be followed by a LTR R15,R15 instruction that tests whether or not the ESTAE setup service executed successfully. Consequently, it is a better habit to change the ESTAE SVCs to SR R15,R15 instead of to NOPR 0. This will successfully fool an LTR instruction. You can change the ESTAEX PC instruction to LA R15,0 to set return code zero.

- It is a good idea to include Z/XDC **VERIFY** commands in your command script to make sure that your zaps do not go off target.

- If you do not have the source listing for your program, or if you do not know where the ESTAE SVCs (etc.) are, then try using Z/XDC's **FIND** command to scan storage for them. Example: **FIND 0A3C MYPROG** searches for X'0A3C' (which is SVC 60, the ESTAE SVC) starting at the entry point of the load module named MYPROG.

- Example: A suitable command script might look like this:

```
MAP MYPROG+0
S Q MYPROG
VER .DOESTAE+24=0A3C
ZAP +0=1BFF
FIND 0A3C HISPROG
FIND
ZAP +0=1BFF
```

- The **MAP** command reads a module map for the MYPROG load module and a csect map (hopefully available) for the csect containing MYPROG's entry address.

- The **SET QUALIFIER** command makes MYPROG's entry csect the default csect. This gives meaning to address expressions that start with a dot. Examples: .+2C, .TOPLOOP and just ".".

- The **VERIFY** command ensures that the location that is X'24' bytes past the DOESTAE label contains an SVC 60 (ESTAE's SVC number). If it does not, then the command script will be aborted.

- The first **ZAP** command changes the SVC 60 to SR R15,R15.

- The first **FIND** command searches a second load module (HISPROG) for an SVC 60. If it does not find one, then the command script will be aborted. Warning, when you are doing the research to create your command script, you must make sure that the **FIND** command does not find data that happens to look like an SVC 60.

- In this example, the programmer discovered that the first **FIND** would find data instead of an actual SVC 60. Therefore, he added a second **FIND,** without operands, to continue searching until a second 0A3C was found. The programmer has previously determined that this second hit is in fact an SVC 60.

- The 2nd **ZAP** then changes that SVC 60 to a SR R15,R15.

Building a command script to suppress your program's error recovery setup process takes a certain amount of effort and research when you first do it. However, once you have created the command script, using it is very easy. You can have Z/XDC execute the command script either automatically or manually:

- MANUALLY: At the start of your debugging session, issue Z/XDC's **READ** command to process the file.

- AUTOMATICALLY: Proceed according to how you are starting your debugging session:

  - If you are starting your debugging session with **XDCCALL** (or any of its aliases), then preallocate your commands dataset to ddname **//XDCCMDS.**

  - If you are starting your debugging session from Z/XDC's Startup Panel in ISPF, then place the name of your commands dataset in the panel field labeled **Commands DSN ===>.**

Help DEbugging EStaes Superseding

z/OS permits any program to issue any number of ESTAE (and similar) macros to create as many ABEND error recovery routines as it likes. Later, if an ABEND occurs, the Operating System gives the newest ESTAE routine control first. Older ESTAEs will receive control only if newer ones fail to request that main program execution be resumed. Thus, newer ESTAE routines supersede older ones.

If a user program issues an ESTAE macro to create its own ESTAE routine and then issues a 2nd ESTAE macro for Z/XDC, then the Z/XDC ESTAE, being newer, will supersede the user program's ESTAE routine, and so Z/XDC will receive control for all ABENDs and allow the programmer to issue debugging commands. The user program's

ESTAE will not receive control unless and until the programmer issues Z/XDC's "END" command which instructs Z/XDC to "percolate" the ABEND. This means that Z/XDC returns a code to z/OS instructing it to proceed with ABEND processing. This causes the Operating System to pass control to the next older ABEND error recovery routine, if any.

A superseding ESTAE for Z/XDC can be setup either by source code changes at assembly time or by Z/XDC's HOOK commands at execution time, or both. Examples:

- **(Source code changes)** Suppose you have modified your program so that it now contains the following:

```
         ESTAE MYESTAE
         ...
XDCGATE  B     XDCZ
         LOAD  EPLOC==CL8'XDC',ERRET=XDCZ   ("==" is not a typo)
         ESTAE (R0)
XDCZ     DS    0H
```

  (If Z/XDC is installed under a clone name ["xxx"], then change =CL8'XDC' to match the clone name ["=CL8'xxx'"]. For information about renaming Z/XDC, see HELP XDCCLONES.)

  If you want to run your program in a debugging session, then start your program using either XDCCALL, or one of its aliases, or Z/XDC's Startup Panel in ISPF. This will cause Z/XDC to receive control before your program even starts. Then you can use Z/XDC's ZAP command to change the instruction at XDCGATE from a branch to a no-op. Then you can set breakpoints at appropriate locations, and then use Z/XDC's GO command to let your program start. Eventually, when execution reaches and executes the "ESTAE MYESTAE" macro, Z/XDC will no longer be usable. But when execution then reaches the LOAD and ESTAE macros for Z/XDC, Z/XDC will be usable again. If execution reaches a breakpoint or an ABEND after that, then Z/XDC will receive control and you can proceed with your debugging session.

  Alternatively, Z/XDC provides a #XDCHOOK macro that can be used in place of the LOAD and ESTAE macros shown above. So using the #XDCHOOK macro, your source code change might be the following:

```
         ESTAE MYESTAE
         ...
XDCGATE  B     XDCZ
         #XDCHOOK ,
XDCZ     DS    0H
```

  One problem, though, with using the #XDCHOOK macro is that it is sensitive to Z/XDC's name. If you typically rename Z/XDC to names other than "XDC", then you will have to reassemble your programs to notify #XDCHOOK of the nonstandard name.

- **(Execution time hooks)** A "dynamic hook" is a special instruction sequence and control block pointer that is temporarily overlaid onto user code at the point at which debugging is to be started. When user execution reaches this special instruction sequence a suitable Z/XDC environment is created and debugging can be commenced. Hooks can be placed by the HOOK and HDEFERRED commands. for more information, see HELP DEBUGGING ESTAES HOOKS.

Help DEbugging EStaes Modifying

Integrating Z/XDC into a Centralized Recovery Routine

If you are working on a large program with its own ABEND error recovery routine, and if you want to create a long-term solution for creating a Z/XDC interface in your program, then probably the most flexible method is to modify your ESTAE[X] to simply call Z/XDC (via a BASR to the **XDC** load module) under appropriate circumstances.

Integrating Z/XDC into your existing ESTAE[X]s is a particularly good solution if your program is designed to incur **"acceptable"** ABENDs that serve useful purposes and for which your ESTAE[X] performs suitable recovery. You would never want Z/XDC to receive control for such "normal" ABENDs. **Examples:**

- A copying program might be susceptible to x37 ABENDs (output dataset/disk is full). The program could have an ESTAE[X] that intercepted the x37 ABEND and reallocated the output file to a larger dataset and then resumed back to the main program to re-perform the copy.

- A storage display program might be susceptible to 0C4 ABENDs if it were directed to display nonexistent or otherwise inaccessible storage. The program could have an ESTAE[X] that intercepted the 0C4 ABEND and resumed back to the main program with a signal that it should issue an error message.

- A formula computation program might be susceptible to various mathematical 0Cx ABENDs such as "divide by 0", overflow, etc. The program could have an ESTAE[X] that intercepted the 0Cx ABEND and resumed back to the main program with a signal that it should issue an error message.

The Structure of a Properly Written Recovery Routine

In general, ESTAE[X] routines can be viewed as having two logical components:

- The first part processes and recovers from **expected** ABENDs.

- The second part processes, records diagnostics about, and either cleans up after or aborts on **unexpected** ABENDs that "should not have occurred".

The best place to code an interface to Z/XDC would be right between these two components.

If your ESTAE[X] does not have a first part (i.e. if in your case all ABENDs are "bad"), then code the Z/XDC interface at the beginning of your ESTAE[X].

Register Inputs When Calling XDC

To make a **normal** call to Z/XDC from your recovery routine, all you have to do is provide to Z/XDC the same register inputs that z/OS’s RTM provided to your recovery routine. Specifically:

R0 - Must not equal 12 (decimal), and it must be nonnegative.

R1 - Must point to the SDWA that was provided to your ESTAE[X] by z/OS.
R2/AR2 - Must either point to a parameter block as mapped by the **#DBCPARM** macro or must be **zero'd.**
R13 - Must point to a standard 72-byte register save area that is available for use by Z/XDC.
R14 - Must point to a return address for Z/XDC to use.
R15 - Must point to the XDC load module's entry point.

R3-R12 - Not used by Z/XDC, but they will be among the registers that are saved and restored.

Conditional and Informational Calls - An Initial Mention

In addition to **normal** calls, Z/XDC also permits its direct callers to make **informational** and **conditional** calls that allow the user's recovery routine to make decisions about whether or not to engage the debugging session based both on the nature of the current abend and on the state of the debugging session itself. All that is discussed in detail below in the section named **Conditional and Informational Calls to Z/XDC.**

For the moment, I'm going to confine my remarks to the register settings needed to distinguish the three call types.

Start by setting R0 as follows:
- **Normal** **Calls:** Just make sure R0 is neither negative nor twelve (decimal).
- **Conditional** **Calls:** Set R0 to **X'DAFE8000'** [DAFE+D0F1COND]
- **Informational** **Calls:** Set R0 to **X'DAFEC000'** [DAFE+D0F1COND+D0F1DOBP]

Notes:

- **X'DAFE'** is a semaphore alerting Z/XDC that a conditional or informational call is being made.

- Notice that **X'DAFE'** make the value in R0 negative; therefore Z/XDC knows that the call is not normal.

- Do not use any other negative value. That would trigger an older interface that still exists but is no longer documented.

- **D0F1COND** and **D0F1DOBP** are flag names created by the **#DBCR0IN** macro.

For all three call types, set **R1, R2** and **R13-R15** to the values described above.

Register Outputs from XDC

Z/XDC will return to the location pointed to by R14. Upon return, Z/XDC will have updated the SDWA appropriately, and it will restore all registers except R0 and R15. R15 will be set to one of the following completion codes:

R15=0 - Your recovery routine has made a **normal call** to Z/XDC, and the user has issued an **END** command. Your recovery routine should proceed as if Z/XDC

had never been called; i.e. it should continue with its normal processing for unwanted ABENDs.

- **R0** contains the return information discussed below in the section named **Conditional and Informational Calls.** It also is described in detail in **HELP * INFORMATIONALCALLS.**

R15=4 - Your recovery routine has made a **normal call** to Z/XDC, and the user has issued a **GO** or **TRACE** command. Your recovery routine must immediately terminate by issuing an **SVC 3** instruction. It **must not** try to make a "clean" return to the Operating System, and it **must not** make any changes to the SDWA.

- **R0:** If your recovery is a **STAE** or **STAI** (ancient types of recovery routines), then **R0** contains information needed for resuming your program, so it should remain unchanged when you issue the SVC 3.
- Otherwise, your program is a more modern ESTAE, ESTAI, etc., and R0 doesn't matter.

R15=-4 - Your recovery routine has made a **conditional** or **informational** call to Z/XDC, which is now returning **with** the requested information but **without** starting or continuing a debugging session. See the **Conditional and Informational Calls** section below for more information.

- **R0** contains the return information discussed below in the section named **Conditional and Informational Calls.** It also is described in detail in **HELP * INFORMATIONALCALLS.**

Notes:

- When BASR'ing to XDC, the call is conditional/informational only you have set **R0** to a **negative** value before calling XDC.
- A **"-4 return"** can occur only for conditional/informational calls. It will **never** occur for normal calls, and it will never occur when Z/XDC is called by the **RTM.**

Using the CALLSXDC Semaphore

When debugging multitasking programs, there are times when Z/XDC needs to determine ahead of time which tasks have ESTAE[X]s or ESTAIs that either are Z/XDC or may call Z/XDC. In other words, Z/XDC needs to know which tasks are part of the debugging session and which are not. (For more information about this, see **HELP MULTITASK SCOPE.)**

Normally, Z/XDC can make this **scope determination** simply by scanning a task's queue of ESTAE[X]s and ESTAIs (specifically, by scanning the task's SCB queue) and seeing whether or not it finds a pointer to the Z/XDC load module.

(You can use the **LIST ESTAES** command to view the **SCB queue** for any task.)

However, when you modify your own ESTAE[X] or ESTAI so that it calls Z/XDC, there is no automatic way that Z/XDC can know that your ESTAE[X] or ESTAI is a "front-end" for Z/XDC.

To solve this problem, if you write a recovery routine that BASRs to **XDC,** then you should place, just ahead of your routine's entry point, the 8-character string: **"CALLSXDC":**

- When searching for an ESTAE[X] or ESTAI routine that might be a part of the debugging session, Z/XDC checks the eight bytes (if any) immediately preceding your recovery routine's **entry point** to see if it contains the character string **CL8'CALLSXDC'.** If so, then Z/XDC knows that your ESTAE[X]/ESTAI may call Z/XDC.

- Note, the **"XDC"** part of this string should **not** be changed even if Z/XDC itself has been renamed to some clone name other than **XDC.**

- See **HELP * EXAMPLE** for an example.

Sometimes You want to Debug, Sometimes You Don't

An additional consideration: It probably would be a good idea to make your recovery routine's interface to Z/XDC **optional;** that is, you probably should structure the interface so that you can externally control whether or not Z/XDC is called even for deservedly bad ABENDs. (It simply would not do, for example, if you shipped a product to a customer, and a debugging session suddenly popped up in front of him.)

Such a controlling option could be implemented in any way that would be appropriate in the context of what your program is and does. Examples:

- You might implement a **DEBUGME** PARM field option that you could use to enable a Z/XDC interface when you start your program.

- You could code your program to search your TIOT for a "keyword ddname" **(DEBUGME** for example). Then if you wanted to enable debugging, you could add a **//DEBUGME DD DUMMY** card to your JCL when you start your program. (See **HELP DDNAMES** for a definition of **Keyword DDNAMEs.)**

- If your program has an interactive command interface (such as **TGETs** or **WTORs** or System Operator **MODIFY** commands), then you could implement a **DEBUGME ON/OFF** command that would allow you to turn your Z/XDC interface on and off at will without having to stop and start your program.

- Perhaps **the simplest,** least elegant, but still very flexible way to implement an optional Z/XDC interface is to code a **zappable gate** around the interface code. The gate can be assembled closed but then zapped open at any appropriate time by any of several methods:

  - If you start your program:
    - Under the control of **XDCCALL** (or XDCCALLA, XDCCMD or XDCCMDA)
    - Or via Option 1 or 2 of the **Z/XDC Startup Panel** in ISPF,

    Then Z/XDC receives control right before your program starts. At that point, you can use Z/XDC's **ZAP** command to open the interface gate.

  - If your program's load module resides permanently in **common storage** (in the PLPA, the MLPA, the FLPA or the DLPA), then you can start an authorized debugging session in TSO against a **token program** (e.g. **TSO XDCCALLA IEFBR14)** and then use Z/XDC's **ZAP** command to open the Z/XDC interface gate. Then when the appropriate address space reaches the interface code, a Z/XDC debugging session starts in that address space. See **HELP DEBUGGING PLPA** for related information.

- If the Z/XDC interface is contained in a long running program running in another address space, you could use Z/XDC's **Foreign Address Space Mode** to access and zap the interface gate. To do so, proceed more or less as follows:
  - In any TSO session, start an authorized debugging session against a token program (example: **TSO XDCCALLA IEFBR14).**
  - Use Z/XDC's **SET ASID** command to target the job, system task, or TSO session in which your program is running.
  - Use the **MAP** command to load a module map and, if available, a csect map for your program.
  - Use the **FORMAT** command to display the code containing the Z/XDC interface gate.
  - Use the **Z** ("Zap") shortcut to open or close the gate, as appropriate.
  - (Optional) Use Z/XDC's **END COMPLETELY** command **(F4)** to end your token debugging session.

  If you opened your recovery routine's Z/XDC interface gate, then do whatever is necessary to cause your program in the other address space to ABEND, thereby causing execution to reach your recovery routine. When it does, your routine will call Z/XDC, and a Z/XDC debugging session will start in that address space.

Then connect to your program's debugging session by whatever method is appropriate (probably via Option **3** of the **z/XDC STARTUP PANEL** in ISPF.)

Conditional and Informational Calls to Z/XDC

Z/XDC supports **three** kinds of direct calls: **normal, informational** and **conditional.** So far, we've discussed mainly the **normal** call...

When a user written ESTAE[X]/ESTAI makes a **"normal"** call to Z/XDC:
- A debugging session is either started or continued.
- The user receives displays and issues commands.
- Z/XDC does not return to your ESTAE[X]/ESTAI until the user issues a **TRACE, GO** or **END** command.
- **R15** will be set to 0 **(END)** or 4 **(TRACE** or **GO).**
- **R0** (for **END** commands) will contain information flags about the abend and about the state of Z/XDC.

But your recovery routine can also make **informational** calls and **conditional** calls to Z/XDC.

An **informational** call does not start or continue a debugging session. Instead, Z/XDC makes an immediate return to your ESTAE[X]/ESTAI with R0 set to certain information about the abend and about the state of the debugging session.

A **conditional** call may or may not start or continue a debugging according to whether or not the current abend is actually a breakpoint...

- For **breakpoints,** the call is converted to a **normal** call, and a debugging session is either started or continued.

- For **other abends,** the call is converted to an **informational** call, and an immediate return is made with **R0** containing information about the abend and about the state of the debugging session.

For full details about conditional and informational calls, see **HELP * INFORMATIONALCALLS.**

Subtopics Index

For more information, select the following topic. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

INFORMATIONALCALLS - Making conditional and informational calls to Z/XDC.
EXAMPLE - A specific example of modifying an ESTAE[X] to interface to Z/XDC.

Help DEbugging EStaes Modifying Informationalcalls

Normal, Conditional and Informational Calls to Z/XDC

Z/XDC call be called directly from a user written ESTAEX or other recovery routine. The reasons why you would do that, and the details of how to do that can be found at **HELP *UP.**

There are **three** kinds of calls that can be made: **normal, informational** and **conditional...**

When a user written ESTAE[X]/ESTAI (etc.) makes a **"normal"** call to Z/XDC:

- A debugging session is either started or continued.

- The user receives displays and issues commands.

- Z/XDC does not return to your ESTAE[X]/ESTAI until the user issues a **TRACE, GO** or **END** command.

- Upon return:
  - R15 will be set to 0 (for **END)** or 4 (for **TRACE** and **GO).**
  - R0 (for **END** commands) will contain information flags about the abend and about the state of Z/XDC.

An **informational** call does not start or continue a debugging session. Instead, Z/XDC makes an immediate return to your ESTAE[X]/ESTAI with R0 set to certain information about the abend and about the state of the debugging session.

A **conditional** call may or may not start or continue a debugging according to whether or not the current abend is actually a breakpoint...

- For **breakpoints,** the call is converted to a **normal** call, and a debugging session is either started or continued.

- For **other abends,** the call is converted to an **informational** call, and an immediate return is made with **R0** containing information about the abend and about the state of the debugging session.

What to Set in R0 (and Other Registers) When Calling Z/XDC

See **HELP *UP** for specific information about what to set up in the registers when making the three different call types to Z/XDC.

Information Returned from Z/XDC via R0

When integrating Z/XDC into an existing recovery routine, it sometimes is useful for your routine to learn some information about the abend and about the state of the debugging session. This knowledge can be used in your logic to decide whether or not to make a normal call for starting or continuing an actual debugging session.

Accordingly, when Z/XDC returns to its caller, **R0 will contain information** about the abend and about the state of the debugging session. This will occur for the following call types:

- **Normal** but only if the user issues an **END** command (not when he issues **TRACE** or **GO)**
- **Informational**
- **Conditional** when it is converted to **informational**

The Details of What gets Returned via R0

When R0 does contain returned information, it will contain the following:

- Information about the category of abend that occurred:
  - The "abend" is actually a **breakpoint** (D0F4BKPT=1)
  - The "abend" is a **dead trap** (D0F4DEAD=1)
  - The abend is an **actual abend** (D0F4BKPT=D0F4DEAD=0)

- Information about the state of the debugging session:
  - A debugging session has **not yet started** (D0F4SESS=1)
  - A debugging session is **already running** (D0F4SESS=0)

- Z/XDC **cannot be used** (D0F4DENY=1)
  Probable reasons include:
  - **END COMPLETELY** previously issued (D0IDENDC)
  - **GO RELEASECAPS** previously issued (D0IDRELC)
  - **Security** issue (D0IDAUTH)
  - **Licensing** issues (D0IDLICF D0IDAUTL D0IDCAPQ)
  - Other reasons are possible.

The above mentioned flag names are fully documented in the **#DBCR0UT** macro. See **HELP MACROS #DBCR0UT** for further details.

The **#DBCR0UT** macro is available in the **hlq.XDCV31.XDCMACS** Product Library. You will have to ask your Systems Programmer for the library's actual name at your Data Center.

Help DEbugging EStaes Modifying Example

The following is an annotated example of a Z/XDC interface that might be coded into an existing ESTAE routine. This code does **not** provide debugging protection to the ESTAE routine itself. Instead, it enhances the protection provided by the ESTAE routine to the user program's main code. (To debug ESTAEs, ESTAIs, SPIEs, and ESPIEs with Z/XDC, see HELP DEBUGGING ESTAES DEBUGGING.)

```
    ****************************************
    * TYPICAL ESTAE ENTRY LINKAGE          *
    ****************************************
1)          DS    0H            MATCH ALIGNMENT
2)          DC    CL8'CALLSXDC' Z/XDC'S SEMAPHORE

3)  MYESTAE  DS    0H
4)          LR    R12,R15       COPY LOCAL BASE
5)          USING MYESTAE,R12   DECLARE IT
6)          CH    R0,=Y(12)     SDWA EXIST?
7)          BE    NOSDWA        NO, SKIP
8)          LR    R10,R1        COPY SDWA PTR
9)          USING SDWA,R10      DECLARE IT

    ****************************************
    * CODE, IF ANY, TO PROCESS "GOOD"      *
    * ABENDS                               *
    ****************************************
            ...
10)         ...
            ...

    ****************************************
    * CONDITIONAL INTERFACE TO XDC         *
    ****************************************
11) XDCGATE  B     XDCDONE       ZAPPABLE GATE

12)         L     R15,X'224'    @'MY ASCB
13)         ICM   R1,15,X'AC'(R15) (ASCBJBNI) BATCH JOBNAME EXIST?
```

```
14)          BNZ   NAMETEST      YES, GO CHECK IT OUT
15)          L     R1,X'B0'(,R15) (ASCBJBNS) GET TSO, STC, OR INIT NAME
16) NAMETEST CLC   0(8,R1),=CL8'MYUID' IS THIS ME?
17) UIDGATE  BNE   XDCDONE       NO, FORGET IT

18)          ICM   R15,15,XDCPTR @'Z/XDC, KNOWN YET?
19)          BNZ   GOTXDC        YES, PROCEED
20)          LOAD  EPLOC==CL8'XDC',ERRET=XDCDONE FIND Z/XDC
21)          ST    R0,XDCPTR     @'Z/XDC
22)          LR    R15,R0        MAKE A COPY

23) GOTXDC   LR    R1,R10        GET @'SDWA
24)          SLR   R0,R0         SET R0≙12
25)          BALR  R14,R15       CALL Z/XDC

26)          LTR   R15,R15       Q: GO/END
27)          BZ    XDCDONE       A: END, PROCEED W/ABEND
28)          SVC   3             A: GO, RESUME USER PROGRAM
29) XDCDONE  DS    0H

    ****************************************
    * CODE TO PROCESS "BAD" ABENDS         *
    ****************************************
             ...
30)          ...
             ...
```

- LINES 1-2
  These lines are important in certain multitasking debugging situations when Z/XDC needs to try to figure out which tasks are part of the debugging session and which are not. When an ESTAE/I calls Z/XDC, the only way that Z/XDC in another task can figure out that this might happen is if that ESTAE/I is immediately preceded by the character string "CALLSXDC". Note, line 1 is needed to ensure that the string immediately abuts to the ESTAE/I's entry point.

- LINES 3-9
  Typical ESTAE entry linkage. It sets up a local base register, and it ensures that an SDWA exists and saves its address. It does not bother saving registers since z/OS does not depend upon them being restored. This leaves the RSA (pointed to by R13) available for use by Z/XDC. If you do want to save registers, then you will have to chain a second RSA for Z/XDC to use.

- LINE 10
  If your ESTAE is designed to handle "expected" ABENDs that can naturally occur in your program's normal course of execution, then that code would go here.

- LINE 11 <XDCGATE B XDCDONE>
  A zappable gate to bypass the Z/XDC interface. To enable the interface, zap this gate from a branch (X'47F0...') to a no-op (X'4700...').

- LINES 12-17
  This is optional code to verify the name of the address space in which the Z/XDC interface code is executing. If the interface is within a program that can be executed by multiple address spaces (such as system exit routines), then this code is necessary so that the Z/XDC interface is enabled **only** for the specific programmer who wants to run a debugging session. It generally would not be appropriate for Z/XDC displays to suddenly appear at the terminals of other

users.

- LINE 12 <L R15,X'224'>
  Storage location X'224' is the system's "PSAAOLD" field. It always points to the Address Space Control Block for the currently executing program.

- LINE 13 <ICM R1,15,X'AC'(R15)>
  Location ASCB+X'AC' is the ASCBJBNI field. If the address space is executing a batch job, then this field points to that job's jobname; otherwise, this field contains a zero.

- LINE 15 <L R1,X'B0'(,R15)>
  If ASCBJBNI does not contain a pointer, then this instruction is executed to load a jobname pointer from the ASCBJBNS field instead, This field is always nonzero and always contains one of the following pointers:
  - For batch jobs: ASCBJBNS points to CL8'INIT'.
  - For system tasks: ASCBJBNS points to the task's procname.
  - For TSO sessions: ASCBJBNS points to the user's userid.

- LINE 16 <NAMETEST CLC 0(8,R1),=CL8'MYUID'>
  The CLC instruction checks the name of the address space within which the interface code is executing.

- LINE 17 <UIDGATE BNE XDCDONE>
  If the interface is running in the "wrong" address space, the "BNE" instruction causes Z/XDC to be bypassed. If you want to disable this check, then zap the BNE (X'4770') to a no-op (X'4700').

- LINES 18-22
  This code locates and saves a pointer to the Z/XDC load module. If the load module cannot be found, then the rest of the interface is bypassed.

- LINE 20 <LOAD EPLOC==CL8'XDC',ERRET=XDCDONE>
  (The "==" is not a typo.) If Z/XDC is installed under a clone name ("xxx"), then change =CL8'XDC' to match the clone name ("=CL8'xxx'"). For information about renaming Z/XDC, see HELP XDCCLONES.

- LINES 23-25
  This code loads the rest of the input registers needed by Z/XDC.

- LINE 24 <SLR R0,R0>
  R0 can be any nonnegative value except 12.

- LINES 26-29
  This code checks and responds to Z/XDC's return code. Z/XDC does not return until the user issues a GO, TRACE, or END command. Then Z/XDC returns with R15=0 if an END command was received, or R15=4 if a GO or TRACE command was received.

- LINE 27 <BZ XDCDONE>
  If R15=0, then the user issued Z/XDC's END command. This means that the user wants to end the debugging session and terminate the program. In this event, the user's ESTAE routine should proceed with its normal processing for "bad" ABENDs.

- LINE 28 <SVC 3>
  If R15≠0, then the user issued a GO or TRACE command. This means that the user wants the main program to resume execution. Accordingly, Z/XDC has already made all appropriate settings in the SDWA, and all that is left to do is to terminate

the ESTAE and return control to z/OS so that it can resume the user's program. So unless the ESTAE routine needs to clean up any of its **own** footprints, it should just issue the SVC 3 to immediately terminate. In any case, the ESTAE must not tamper with the SDWA.

- LINE 30
  Code should be placed here to handle unexpected or "bad" ABENDs. Normally, this code would release/restore critical resources, record problem related information, and/or terminate at least the failing parts of the user program.

Help DEbugging EStaes Espies

SPIEs and ESPIEs are exit routines that a user program can create to intercept a category of ABENDs called "program checks". Program checks are machine instruction coding or data errors that, if not intercepted, eventually lead to 0Cx ABENDs. Program checks (also known as "program interrupts") are described in the various "Principles of Operation" manuals, including "z/Architecture Principles of Operations", SA22-7832. 0Cx ABENDs are described in "MVS System Codes", (SA22-7626 for z/OS, GC28-1780 for OS/390).

Briefly, SPIE and ESPIE routines can intercept only program check type ABENDs. All other types of ABENDs are processed by ESTAE/ESTAI type routines. SPIE and ESPIE routines can be selective about which program check ABENDs they will and will not intercept. If a program check is not intercepted by a SPIE/ESPIE, then it will be forwarded to ESTAE/ESTAI processing. For more information about SPIE/ESPIE routines, see the "Program Interruption Services" chapter of "MVS Programming: Assembler Services Guide", GC28-1762.

If a user program creates a SPIE or ESPIE routine to intercept one or more program checks, then when the program check occurs, z/OS will always pass control to the SPIE/ESPIE first, ahead of any ESTAE/ESTAI type routine. Therefore, SPIE/ESPIE routines can interfere with a Z/XDC debugging session.

- If your program has a SPIE/ESPIE, then it is *essential* that it not intercept s0C1 ABENDs (invalid operation). This is because Z/XDC breakpoints are implemented via s0C1 ABENDs. Thus, if a breakpoint is executed, then the SPIE/ESPIE would receive control instead of Z/XDC.

- It also is strongly recommended that a user SPIE/ESPIE not intercept at least the rest of the nonmathematical program check ABENDs: 0C2-0C7. This is because these ABENDs usually occur from coding errors that need to be debugged rather than data errors that just need to be handled.

- It usually is OK for a SPIE/ESPIE to intercept the mathematical program checks. This is because they usually result from invalid data instead of from coding logic errors.

If your program has a SPIE or ESPIE routine that does not conform to the above limitations, then you must either change it or remove it from your program before you will be able to use Z/XDC. This can be done either by source code changes at assembly time or by ZAP commands at execution time, or by use of Z/XDC's HOOK command. Using the HOOK command is probably the easiest. (See the following topic: HELP DEBUGGING ESTAES HOOKS.)

To disable your SPIE/ESPIE routines at execution time, start your debugging session using XDCCALL (or any of its aliases or the Z/XDC Startup Panel in ISPF). Display the code that you want to debug, and place an initial trap into it using the HOOK command. Then use the GO command to begin execution. When your program reaches the hook, SPIE/ESPIE processing will be suppressed for program checks 1 through 7, a Z/XDC environment will be created, and Z/XDC will be given control to display your code and accept commands.

RECOMMENDATION: z/OS's overhead for SPIE/ESPIE routines is quite a bit less than it is for ESTAE/ESTAI routines; however, unless your program is *constantly* receiving and recovering from program check ABENDs, the processing savings of using SPIE/ESPIEs is not going to amount to more than a few seconds per year! Furthermore, both the execution environments supported and the recovery options available are significantly greater for ESTAE/ESTAIs than they are for SPIE/ESPIEs. Because of this, and because SPIE/ESPIEs create greater interference with Z/XDC than do ESTAE/ESTAIs, I would like to recommend that you do away with your SPIE/ESPIEs and implement the equivalent support into your ESTAE/ESTAIs instead.

Help DEbugging EStaes Hooks

If you have a long term commitment to the development and debugging of a complex program, then probably the best way to use Z/XDC would be to integrate Z/XDC into that program by modifying its ESTAE/ESTAIs to call Z/XDC under suitable conditions. (See HELP DEBUGGING ESTAES MODIFYING for detailed information.)

On the other hand, in many situations the HOOK command provides a very powerful and easy to use method for passing control to Z/XDC regardless of a piece of code's recovery environment.

The HOOK command can target code running under the current task, under a different task, or even in a different address space. When the targeted code executes the trap that has been created by the HOOK command, a Z/XDC environment will be built, and Z/XDC will receive control, and a debugging session will either start or continue.

The HOOK command sets a trap by overwriting 6 bytes of the targeted code with a JLU instruction that jumps to the Hook Service's Prolog code. When execution in the targeted code reaches the Hook Point, the JLU is removed, user code is restored, and a Z/XDC environment is created as follows:

- If a prior ESTAE-type recovery routines exist (ESTAEX, ESTAI, ARR, etc.), then they are all superseded by setting Z/XDC up as an ESTAEX that is newest of all.

- If prior FRRs exist, then they are all superseded by setting Z/XDC up as a the newest FRR.

- If any SPIEs or ESPIEs exist, then they are defeated by finding the IBM control block that defines the existence of the SPIE/ESPIE and changing its mask to prevent the interception of program checks 1 through 7 (s0C1-s0C7).

Once a hook has been executed, debugging can proceed using Z/XDC's normal breakpointing (and other) commands. The environment created by a hook remains valid until the user program takes some action to change the environment. Such actions include:
- Creating a newer ESTAE or ESTAEX routine.

- Issuing a **GO REMOVEXDC** command. This removes the ESTAEX or FRR that was created by the HOOK. See HELP COMMANDS GO for more information.
- ATTACH'ing a subtask. The debugging environment will not automatically propagate to the subtask. If you want to debug the subtask, then you will have to place another hook into the subtask's code in order to do so.
- Running a program or exit routine under a newer Request Block (RB). Said routine will not run under the debugging environment created by the hook. Such routines include:
  - A program invoked via LINK.
  - An OPEN exit routine (DCB exit, EOV exit, etc.).
  - A STIMER exit.
  - An ETXR exit.
  - An attention exit.
  - Any other IRB routine.
  - An ESTAE or ESTAI exit.

For more information, please see the following topics:

HELP HOOKS (and subsequent)

HELP COMMANDS HOOK

HELP COMMANDS HDEFERRED

Help DEbugging EStaes Debugging

Z/XDC can be used to debug both other ABEND error recovery routines (ESTAEs, ESTAIs, etc.) and program check intercept routines (SPIEs and ESPIEs).

SPIEs and ESPIEs: No special action needs to be taken for debugging program check intercept routines. As long as Z/XDC is properly setup to debug the routine in which an intercepted program check occurs, Z/XDC can directly debug the SPIE/ESPIE routine that intercepted that program check. Just set breakpoints at the appropriate locations within your SPIE/ESPIE, and then run your program, doing whatever is necessary to create the program check that will be intercepted by your SPIE/ESPIE routine.

ESTAEs, ESTAIs, etc.: If you start your debugging session using XDCCALL (or any of its aliases) or using Z/XDC's Startup Panel in ISPF, then your program will be ATTACH'd as a subtask with Z/XDC established as the subtask's ESTAI routine. If your program then issues an ESTAE macro and then ABENDs, then your ESTAE will receive control. If you ESTAE then ABENDs or reaches a breakpoint, Z/XDC will then receive control and you will be able to debug your ESTAE.

The situation becomes more complex if your program has multiple ESTAE/ESTAIs or if you start your debugging session without the aid of XDCCALL (et al.). In these situations, if you want to debug your ESTAE/ESTAI routine with Z/XDC, then you may have to issue an ESTAE macro from within your ESTAE/ESTAI making Z/XDC the ESTAE routine for your ESTAE/ESTAI. This is the same method needed for debugging most other exit routines. See HELP DEBUGGING EXITROUTINES for more information and examples.

To understand fully how a Z/XDC ESTAE/ESTAI routine interacts with other ABEND error recovery routines and with programs making use of multiple Request Blocks in general, please see HELP EXECUTIONLEVELS.

Help DEbugging EXitroutines

Z/XDC can be used to debug just about any kind of exit routine that runs either in a task mode or an SRB mode environment. This includes the following nonexhaustive list:

- Product exits:
  TSO SUBMIT command exits
  IOF exits
  SDSF exits

- System component exits:
  SMF exits
  DASDM exits
  TSO logon exits
  JES2 exits
  JES3 exits
  SAF exits
  RACF exits
  CA-ACF2 exits
  Exits of other products that run as system components

- Programming exits that run under their own Request Blocks:
  DCB exits (OPEN, LABEL, EOV, etc.)
  STIMER exits
  Attention (STAX) exits
  VTAM IRB exits
  ETXR exits
  Certain subsystem exits (take care!)

- PC routines

- SRB routines

- (Note: for debugging SPIE, ESPIE, ESTAE, ESTAI, and other ABEND error recovery exits, see HELP DEBUGGING ESTAES DEBUGGING.)

In general, the above lists represent two broad categories of exit environments:

- Those product and system exits that generally run within "black box" environments that already have their own established ABEND error recovery routines which interfere with an externally created Z/XDC ESTAI routine (such as that created by the XDCCALL program). Consequently, the exit routine has to create its own ESTAE routine in order to make use of Z/XDC debugging.

- Those programming exits that run under the control of Request Blocks that are different from the RBs under which their main programs run.

  There exists an "ownership" relationship between RBs and ESTAE/ESTAIs (such as Z/XDC) that defines rules governing whether or not a program running under a given RB can be resumed by a given ESTAE/ESTAI routine after an ABEND has occurred. For complete information about this, see HELP EXECUTIONLEVELS. Meanwhile, suffice it to say that in order for an ESTAE routine to be able to

resume the execution of a given programming exit, it is required by z/OS that the exit be the "owner" of that ESTAE. This means that the exit has to have created the ESTAE by having issued an ESTAE (or similar) macro.

However, for all these different kinds of exits, basically the same technique can be used to insert a Z/XDC interface into them:

- For some product exits (such as CICS) you may have to issue an "ESPIE RESET" macro to cancel a SPIE or ESPIE routine.

- Locate a copy of the Z/XDC load module (sometimes via a LOAD macro, sometimes via a saved pointer to a previously LOAD'd copy, depending upon the specifics of your exit's environment).

- Issue an ESTAE or ESTAEX macro making Z/XDC the current environment's newest ABEND error recovery routine.

- Optionally, issue a #DIE macro to pass control immediately to Z/XDC so that further breakpoints can be set.

An additional consideration: It is a good idea to make your Z/XDC interface conditional; that is, you should structure the interface so that you can externally control whether or not Z/XDC is called when your exit routine is executed.

Perhaps the simplest, least elegant, but still very flexible way to implement a conditional Z/XDC interface is to code a zappable gate around the interface code. Usually, you will want to assemble the gate closed. You then can zap it open at any appropriate time by any of several methods:

- If your exit runs in a program that you can start under the control of XDCCALL (or any of its aliases or the Z/XDC Startup Panel in ISPF), then Z/XDC receives control right before your program starts. At that point, you can use Z/XDC's ZAP command to open your exit's Z/XDC interface gate.

- If your exit routine resides permanently in common storage (in the PLPA, the MLPA, the FLPA or the DLPA), then you can start an authorized debugging session against an arbitrary program (e.g. "XDCCALLA IEFBR14"), and then use Z/XDC's ZAP command to open the Z/XDC interface gate. Then when the appropriate address space executes the interface code, a Z/XDC debugging session starts in that address space. See HELP DEBUGGING PLPA for related information.

- If your exit routine is contained in a long running program running in another address space, you could use Z/XDC's Foreign Address Space Mode to access and zap the interface gate. To do so, proceed more or less as follows:
  - Start an authorized debugging session against an arbitrary program (example: "XDCCALLA IEFBR14").
  - Use Z/XDC's SET ASID command to target the job, system task, or TSO session in which your program is running.
  - Use the MAP command to load a module map and, if available, a csect map for your program.
  - Use the FORMAT command to display the code containing the Z/XDC interface gate.
  - Use the **Z** ("Zap") shortcut to open or close the gate, as appropriate.
  - If you opened the interface gate, then do whatever is necessary to cause the program in the other address space to call the exit. When it does, a Z/XDC debugging session will start in that address space.

- Use Z/XDC's END COMPLETELY command to end your arbitrary debugging session.
- Then connect to your program's debugging session by whatever method is appropriate (probably via cdf/XDC).

Subtopics Index

For more information, select the following topic. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

EXAMPLES - Specific examples of Z/XDC interface code for exit and other routines.

Help DEbugging EXitroutines Examples

The following are annotated examples of various kinds of Z/XDC interfaces that might be coded into the beginning of different kinds of exit routines.

Programming exits are exit routines that run as part of your program but run under the control of their own Request Blocks. Examples of programming exits include: DCB exits (OPEN, LABEL, EOV, etc.), STIMER exits, Attention (STAX) exits, VTAM IRB exits, and ETXR exits.

Generally, the following very simple Z/XDC interface is sufficient for programming exits.

```
    ************************************
    * SUBROUTINE ENTRY LINKAGE: SAVE     *
    * REGISTERS, ESTABLISH LOCAL         *
    * ADDRESSABILITY, OBTAIN LOCAL WORK  *
    * AREAS, ETC.                        *
    ************************************
    MYEXIT   DS    0H
             ...
1)           ...
             ...

    ************************************
    * Z/XDC INTERFACE SETUP CODE        *
    ************************************
2)           #XDCIS ANSWER=XDCNAME
3)           LOAD  EPLOC=XDCNAME,ERRET=XDCDONE FIND Z/XDC
4)           NILH  R0,X'7FFF'
5)           SYSSTATE ASCENV=P
6)           ESTAEX (R0)
7)           #DIE  'HIYA, KIDS. HIYA, HIYA!'

    ************************************
    * THE REST OF THE EXIT               *
    ************************************
8)  XDCDONE  DS    0H
             ...
             ...
```

```
            ...
```

- **LINE 1**
  Generally, the Z/XDC interface setup code should be placed just following your exit or subroutine's entry linkage.

- **LINE 2 <#XDCIS ANSWER=XDCNAME>**
  This macro determines Z/XDC's current name. (It might be "XDC", or it might be some other 3-character name. See HELP XDCCLONES for more information.)

- **LINE 3 <LOAD EPLOC=XDCNAME,ERRET=XDCDONE>**
  The LOAD macro returns Z/XDC's address in R0.

  The LOAD macro causes an SVC to be issued, so if your exit routine is a PC routine, or is running in a cross memory environment, or is running in access register mode (ASC-MODE=AR), or is running under some other SVC incompatibility, then you will not be able to use the LOAD macro. In that case, you will have to have code located elsewhere that pre-LOADs the XDC load module into storage and then save its address somewhere for this routine to find and load into R0. Example: "L R0,XDCPTR".

- **LINE 4 <NILH R0,X'7FFF'>**
  Now here's a good habit to get into: Always clear the hi-bit (the AMODE flag) from the returned address prior to using it. You never know when you might make the mistake of trying to use the pointer while running in 64-bit addressing mode (AMODE-64). If you try to use the address with the high bit on and while in AMODE(64), you are highly likely to get an instant 0C4 ABEND.

- **LINE 5 <SYSSTATE ASCENV=P or =AR>**
  Issue this macro, as appropriate and as needed, to affect the code generated by the ESTAEX macro.

- **LINE 6 <ESTAEX (R0)>**
  The standard form of the ESTAEX macro is nonreentrant. If your code must be reentrant, then you will have to use an "MF=E" form here and create an MF=L form elsewhere in an available data area.

- **LINE 7 <#DIE 'HIYA, KIDS. HIYA HIYA!'>**
  The #DIE macro is available in the XDCMACS library. The library's factory default name is hlq.XDCV31.XDCMACS. Ask your Systems Programmer for its actual name at your Data Center.

  #DIE creates an assembled breakpoint for Z/XDC, so it is a good way to "call" Z/XDC from a program. See HELP BREAKPOINTS DEADTRAPS for more information.

  Generally, for programming exits you may not want to bother with a #DIE macro. This is because you will probably have opportunities to set breakpoints into the exit routine either at the start of your debugging session or while debugging other parts of your program.

When a programming exit "returns", the Request Block under which it was running is purged by z/OS. All recovery routines that are owned by such RBs also are purged. Therefore, there is no need to worry about canceling the Z/XDC ESTAEX when your exit completes.

- - -

System exits (SMF, DADSM, etc.) and to a certain extent product exits as well require a more complex Z/XDC interface. There can be several reasons for this. One in particular is that SYSTEM exits can be executed from multiple address spaces simultaneously. Therefore, great care must be taken to ensure that a Z/XDC interface and environment affects **only** the address space in which you are running your debugging session. For important related information see HELP DEBUGGING PLPA.

The following is an elaborate example of a Z/XDC interface that might be coded into the beginning of a system exit or a product exit routine.

```
    ***************************************
    * SUBROUTINE ENTRY LINKAGE: SAVE      *
    * REGISTERS, ESTABLISH LOCAL          *
    * ADDRESSABILITY, OBTAIN LOCAL WORK   *
    * AREAS, ETC.                         *
    ***************************************
    MYEXIT   DS    0H
             ...
1)           ...
             ...

    ***************************************
    * CONDITIONAL INTERFACE SET-UP CODE   *
    * GATE                                *
    ***************************************
2)  XDCGATE  B     XDCDONE       ZAPPABLE GATE

    ***************************************
    * ADDRESS SPACE CHECK GATE            *
    ***************************************
3)           L     R15,X'224'    @'MY ASCB
4)           ICM   R1,15,X'AC'(R15) (ASCBJBNI) BATCH JOBNAME EXIST?
5)           BNZ   NAMETEST      YES, GO CHECK IT OUT
6)           L     R1,X'B0'(,R15) (ASCBJBNS) GET TSO, STC, OR INIT NAME
7)  NAMETEST CLC   0(8,R1),=CL8'MYUID' IS THIS ME?
8)  UIDGATE  BNE   XDCDONE       NO, FORGET IT

    ***************************************
    * LOCATE A COPY OF Z/XDC              *
    ***************************************
9)           #XDCIS ANSWER=XDCNAME
10)          LOAD  EPLOC=XDCNAME,ERRET=XDCDONE FIND Z/XDC
11)          LR    R2,R0         COPY @XDC
12)          LA    R2,0(,R2)     PURIFY

    ***************************************
    * CHECK TO SEE IF Z/XDC ALREADY IS    *
    * MY ESTAEX                           *
    ***************************************
13)          L     R1,X'21C'     @MY TCB
14)          ICM   R1,7,X'A1'(R1) @NEWEST SCB; EXIST?
15)          BZ    XDCESTAE      NO, GO ISSUE ESTAEX
16)          L     R1,4(,R1)     @NEWEST ESTAEX EXIT
17)          LA    R1,0(,R1)     PURIFY
18)          CLR   R2,R1         IS Z/XDC ALREADY ESTABLISHED?
19)          BNE   XDCESTAE      NO, GO ISSUE ESTAEX
20)          DELETE EPLOC==CL8'XDC' YES CANCEL EXCESS "LOAD"
```

```
21)          B     XDCBPT        GO TRAP TO Z/XDC

    ************************************
    * REENTRANT CODE TO MAKE IT MY NEWEST *
    * ESTAEX                           *
    ************************************
22) XDCESTAE GETMAIN RC,LV=24    GET PLIST SPACE
23)          LTR   R15,R15       GOT 24 BYTES?
24)          BNZ   XDCDONE       NO, FORGET IT.
25)          LR    R3,R1         SAVE PLIST PTR.
26)          XC    0(24,R3),0(R3) CLEAR IT
27)          SYSSTATE ASCENV=P
28)          ESTAEX (R2),CT,PURGE=NONE,ASYNCH=YES,TERM=YES,MF=(E,(R3))
29)          FREEMAIN RU,A=(R3),LV=24 DONE W/ESTAEX PLIST

    ************************************
    * ASSEMBLED BREAK-POINT            *
    ************************************
30) XDCBPT   #DIE  'HIYA, KIDS. HIYA, HIYA!'

    ************************************
    * THE REST OF THE EXIT/SUBROUTINE     *
    ************************************
31) XDCDONE  DS    0H
             ...
             ...
             ...
```

- **LINE 1**
  Generally, the Z/XDC interface setup code should be placed just following your exit or subroutine's entry linkage.

- **LINE 2 <XDCGATE B XDCDONE>**
  This is a zappable gate to bypass the Z/XDC interface. To enable the interface, zap this gate from a branch (X'47F0...') to a no-op (X'4700...').

- **LINES 3-8**
  This is optional code to verify the name of the address space in which the Z/XDC interface code is executing. If the interface is within a program that can be executed by multiple address spaces (such as system exit routines), then this code is necessary so that the Z/XDC interface is enabled **only** for the specific programmer who wants to run a debugging session. It generally would not be appropriate for Z/XDC displays to suddenly appear at the terminals of other users.

- **LINE 3 <L R15,X'224'>**
  storage location X'224' is the system's "PSAAOLD" field. It always points to the Address Space Control Block for the currently executing program.

- **LINE 4 <ICM R1,15,X'AC'(R15)>**
  Location ASCB+X'AC' is the ASCBJBNI field. If the address space is executing a batch job, then this field points to that job's jobname; otherwise, this field contains a zero.

- **LINE 6 <L R1,X'B0'(,R15)>**
  If ASCBJBNI does not contain a pointer, then this instruction is executed to load a jobname pointer from the ASCBJBNS field instead, This field is always

nonzero and always contains one of the following pointers:
- For batch jobs: ASCBJBNS points to CL8'INIT'.
- For system tasks: ASCBJBNS points to the task's procname.
- For TSO sessions: ASCBJBNS points to the user's userid.

- LINE 7 <NAMETEST CLC 0(8,R1),=CL8'MYUID'>
The CLC instruction checks the name of the address space within which the interface code is executing.

- LINE 8 <UIDGATE BNE XDCDONE>
If the interface is running in the "wrong" address space, the "BNE" instruction causes Z/XDC to be bypassed. If you want to disable this check, then zap the BNE (X'4770') to a no-op (X'4700').

- LINES 9-12
This code locates and saves a purified pointer to the XDC load module. If the load module cannot be found, then the rest of the interface is bypassed.

- LINE 9 <#XDCIS ANSWER=XDCNAME>
This macro determines Z/XDC's current name. (It might be "XDC", or it might be some other 3-character name. See HELP XDCCLONES for more information.)

- LINE 10 <LOAD EPLOC=XDCNAME,ERRET=XDCDONE>
The LOAD macro returns Z/XDC's address in R0.

The LOAD macro causes an SVC to be issued, so if your exit routine is a PC routine, or is running in a cross memory environment, or is running in access register mode (ASC-MODE=AR), or is running under some other SVC incompatibility, then you will not be able to use the LOAD macro. In that case, you will have to have code located elsewhere that pre-LOADs the XDC load module into storage and then save its address somewhere for this routine to find and load into R0. Example: "L R0,XDCPTR".

- LINE 12 <LA R2,0(,R2)>
The address returned for Z/XDC needs to be purified because the hi-order bit may be on indicating that the System believes that Z/XDC needs to be called in amode-31. Actually, Z/XDC can be called either in amode-31 or amode-24, so this bit is not needed.

- LINES 13-21
This is a fancy piece of code that checks to see whether or not Z/XDC already is the newest ESTAEX in the current environment. If so, then there is no need to issue an ESTAEX again.

- LINE 13 <L R1,X'21C'>
Storage location X'21C' is the system's "PSATOLD" field. It always points to the Task Control Block for the currently executing program.

- LINE 14 <ICM R1,7,X'A1'(R1)>
Location TCB+X'A1' is the TCBSTABB field. It contains the 24-bit address of the newest STAE Control Block (SCB) defined for the current task. SCBs are used by z/OS to identify and control ABEND error recovery exit routines.

Note, prior to the ICM instruction, R1 contained the address of the current TCB which is guaranteed to be a 24-bit address; therefore, there is no need to clear R1's hi-byte for the ICM.

- **LINE 15 <BZ XDCESTAE>**
  If no SCBs are queued to TCBSTABB, then no ESTAEXs have been created, so Z/XDC definitely is not the environment's newest ESTAEX, so it is necessary to go issue an ESTAEX macro for Z/XDC.

- **LINE 16 <L R1,4(,R1)>**
  Location SCB+4 is the SCBEXIT field. It points to the recovery routine that is defined by the SCB. Therefore, this instruction causes R1 to point to the current task's newest recovery exit routine.

- **LINE 17 <LA R1,0(,R1)>**
  This clears a possible "amode-31" flag.

- **LINE 18 <CLR R2,R1>**
  This checks to see if the newest recovery routine is, in fact, Z/XDC.

- **LINE 19 <BNE XDCESTAE>**
  If not, then I need to go issue an ESTAEX macro making Z/XDC the newest ESTAEX.

- **LINE 20 <DELETE EPLOC==CL8'XDC'>**
  If so, then the LOAD macro issued above has incremented Z/XDC's load module "use count", so this "DELETE" decrements the use count back down to what it was before the LOAD was issued; it will **not** delete Z/XDC from storage.

- **LINE 21 <B XDCBPT>**
  This skips the ESTAEX macro and jumps directly to the assembled Z/XDC breakpoint.

- **LINES 22-29**
  This is reentrant code that issues an ESTAEX macro making Z/XDC the newest ABEND error recovery routine for the current task. Basically, this code GETMAINs 24 bytes in which a parameter list is built for the ESTAEX macro. Then the ESTAEX macro is issued, and then the plist is FREEMAIN'd. Notes:

  - If the exit in which your are coding this interface is not reentrant, then this code does not have to be reentrant either, and so this code can then be substantially simplified.

  - But even if your exit routine is reentrant, if Z/XDC itself is in the PLPA, then this code does not have to be reentrant, since all users of the code would obtain the same address for Z/XDC.

  - On the other hand, if this code itself is located in the PLPA (or in some other store protected area of storage), then the code **must** be reentrant; otherwise, an 0C4 will occur when the ESTAEX macro tries to store Z/XDC's address into its plist.

  - Finally, if your routine's entry linkage already obtains a storable work area, then you could build the ESTAEX plist there instead of having to GETMAIN/FREEMAIN a plist here.

- **LINE 30 <XDCBPT #DIE 'HIYA, ...'>**
  The #DIE macro is available in the XDCMACS library. The library's factory default name is hlq.XDCV31.XDCMACS. Ask your Systems Programmer for its actual name at your Data Center.

  #DIE creates an assembled breakpoint for Z/XDC, so it is a way to "call" Z/XDC

from a program. See HELP BREAKPOINTS DEADTRAPS for more information.

- Note: You may want to add code that sets a flag when you issue the ESTAEX so that your exit can then issue an "ESTAEX 0" macro before returning to its caller. But the practical need for this depends upon the specific environment within which your exit runs. It is my experience that this is not usually necessary.

Help DEbugging Frr

Z/XDC can run as any type of error recovery routine, including as an FRR routine. Just set up Z/XDC as your program's newest FRR. Do this in the same way that you would for your own FRR, but designate Z/XDC's normal entry address in place of your own FRR routine.

Z/XDC's normal entry point usually is named **XDC,** but when Z/XDC is installed under a clone name ("V31" for example), then its entry point name will be the clone name. (For information about renaming Z/XDC, see HELP XDCCLONES.)

IMPORTANT: Unlike the case for ESTAE-type recovery routines, when Z/XDC is used as an FRR, it is not possible for it to be invoked as a subroutine by an existing FRR. Instead, Z/XDC must replace your existing FRR.

In almost all cases when Z/XDC is intended to be used as an FRR, its **XDC** and **XDC31** load modules must be installed into common storage (the PLPA or on the Dynamic Link Pack Queue).

When the **XDC** load module resides in common storage, you can use our **#XDCLOC8** macro to find its entry address. (This is called Z/XDC's **Quick Locate** support.) In constrained environments, where use of z/OS's LOAD macro is prohibited, the #XDCLOC8 macro can be used instead. For more information, see HELP MACROS #XDCLOC8.

When running as an FRR, Z/XDC can be used to debug both task mode programs and SRB mode routines. For debugging task mode code, use the SETFRR macro to setup Z/XDC as your code's FRR routine. For information specific to debugging SRB routines, see HELP DEBUGGING SRBMODE.

Inherent Dangers

In z/OS, environments in which FRRs would be used can be both highly constrained and highly sensitive. There will be circumstances where interactive debugging can be conducted safely, and you will find Z/XDC to be an extremely valuable tool there. But there also will be environments where use of Z/XDC will simply be **too dangerous** or not even possible! So you will have to learn from experience when Z/XDC can be used and when it can't.

Because of the inherent dangers involved in FRR protected environments, it is highly recommended that it be attempted only in sandbox systems (development systems specifically designated for the testing of risky code). Using Z/XDC as an FRR must **never** be attempted on production systems!

Perhaps the most dangerous aspect of debugging FRR protected code (task mode or SRB

mode) is the requirement that all locks that might have been held by the target code have to be released prior to Z/XDC being able to produce its displays and receive commands. This issue is discussed in detail in HELP DEBUGGING LOSTLOCKS.

In order to allow a data center to control who may and may not be allowed to make use of this risky capability (and on which systems), Z/XDC checks a security rule named **XDC.LOSTLOCKS.** See HELP SECURITY LOSTLOCKS for more information about this.

FRR Proxy Task

Z/XDC's normal processing requires it to use a host of z/OS services that are not compatible with FRR protected environments. These services include such things as allocating and using datasets, communicating with user terminals, waiting for responses, etc. All of these services (and many more) must occur within an unlocked task mode environment where no FRR exists. They cannot occur within an SRB routine; they cannot occur within a task for which an FRR has been established; and they cannot occur while locks are held.

Consequently, the bulk of Z/XDC processing must occur outside of an FRR protected environment. (This is true regardless of whether the FRR is protecting a task mode routine or an SRB mode routine.) So in order to do most of the processing necessary for interactive debugging, an instance must be created of Z/XDC running elsewhere, under a task not affected by an FRR routine. This is done by finding a suitable task within the current home address space and scheduling an IRB for running Z/XDC in task mode.

So for FRR protected routines, Z/XDC's processing splits into two phases: a **Local** Phase and a **Remote** Phase:

- The Local Phase is the FRR code. It receives control when an ABEND (or breakpoint) occurs, and it runs in the highly constrained FRR environment.
- The Remote Phase is the bulk of Z/XDC processing. It runs in a separate task where it is free to do all the things that cannot be done in an FRR environment.

The task that the Remote Phase runs under is referred to as a **Proxy Task.** Whenever a Proxy Task is in control:

- The phrase **SRB PROXY** or **FRR PROXY** is displayed in the upper left-hand corner of your terminal's display screen.
- The **LIST TASKS** command labels the Proxy Task with the phrase **SRB-FRR PROXY** or **TCB-FRR PROXY.**

Z/XDC searches pretty much all tasks when looking for an eligible Proxy Task. However, some tasks are better candidates (for running the Remote Phase) than others. The candidates are (in order starting with the best candidates):

- **Formal** (i.e. intentional) Proxy Tasks (see below) occurring within the scope of a debugging session,
- Formal Proxy Tasks occurring outside the scope of a debugging session,
- The task that owns the debugging session (displayed by **LIST TASKS** as the highest shown hilighted task),
- Tasks contained within the scope of the debugging session (again as hilighted by the LIST TASKS command),
- Tasks outside the scope of the debugging session.

So when searching for a task to use as a Proxy Task, Z/XDC's Local Phase will search tasks in the order listed above, and it will use the first one that seems suitable.

Once a **suitable** task is chosen, Z/XDC's Local Phase schedules the Remote Phase to run as an **IRB** under that task's TCB.

When checking to see whether or not a task is **suitable,** Z/XDC is concerned mainly with whether or not an IRB can be immediately scheduled to run. (If not, then the Local Phase discards the task as a candidate and moves on to the next.) So the kinds of things the Local Phase checks for are:

- Whether IRBs are currently being delayed (TCBFX=1 or TCBNOIRB=1),
- Whether the TCB is in the process of terminating (TCBDYING=1),
- Whether the task is suspended (TCBXLAS<>0),
- Whether an EUT FRR is currently defined for the task (TCBNSSP<>0),
- Whether the task is suspended due to a Set Must Complete ENQ in effect (TCBSMC=1),
- Whether the task is already being used as a Proxy Task,
- And the like.

If Z/XDC's Local Phase cannot find any tasks that are suitable, then it will attempt to ATTACH one. If it cannot do that, then it will give up and abort at a DEAD trap containing error message **DBC902T.**

Creating Formal Proxy Tasks

Unfortunately, Z/XDC cannot predict the future with precision, so left to its own devices, it could easily find a task that it deems suitable that, due the the program's internal logic, actually is not suitable. (Typically, this can lead to deadlocks or other unpleasantnesses.)

To help avoid such problems, Z/XDC offers a couple of ways for you to create **intentional** Proxy Tasks (also called **Formal Proxy Tasks)** where Z/XDC's Remote Phase can run without unnecessarily interfering with the program being debugged...

- If you have started your debugging session using the **xxxCALLA** utility, then you can add a **//xxxFPTnn DD DUMMY** to your JCL (or use the ALLOCATE command if the program you are debugging is running in TSO). This is a signal that causes xxxCALLA to ATTACH **nn** Formal Proxy Tasks prior to starting your program. The Proxy Tasks will be created as daughter tasks of the xxxCALLA task and sisters to the task in which your program will be ATTACH'd. See HELP DDNAMES FPTNN for more information.

  (Note, by default, absent a //xxxFPTnn allocation, xxxCALLA will automatically create **2** Proxy Tasks.)

- From within any authorized debugging session (whether started by XDCCALLA or otherwise), you can use the following commands:

  - **SET PROXYTASKS** can be used to create Formal Proxy Tasks as subtasks of any suitable TCB you choose.

  - **DELETE PROXYTASKS** can be used to delete Formal Proxy Tasks.

  - **LIST TASKS** can be used to display what Proxy Tasks exist and what their current usage states are.

- From an external debugging session (i.e. one running in another address space),

you can go into Z/XDC's Foreign Address Space Mode and start Formal Proxy Tasks as subtasks of any TCB you choose. Example:

```
XDCCALLA IEFBR14
    SET ASID MYJOB
    LIST TASKS
    SET PROXYTASKS TCB#n
```

Summary of Processing

When Z/XDC receives control as an FRR, the Local Phase proceeds as follows:

- It saves a copy of the SDWA.
- It sets flags in the original SDWA to cause the release of **all** locks that might be held by the protected routine. (This is necessary because of the waiting that the Local Phase is going have to do and because of the processing that the Remote Phase is going to have to do.)
- It then exits the FRR environment in favor of a retry routine. (This is necessitated by constraints within RTM with respect to the handling of SDWA's.) Note that once in the retry routine environment, all locks will have been released.
- The Local Phase (using the process described above) attempts to find a task that it deems suitable to use as a Proxy Task.
- If no suitable task (formal or otherwise) can be found, then it attempts to ATTACH a new Formal Proxy Task.
- If it cannot do this, then it all falls apart, and the Local Phase is reABENDed at a DBC902T DEAD trap.)
- On the other hand, if a suitable Proxy Task is either found or created, then the Local Phase schedules the Remote Phase to run as an IRB within that task.
- The Local Phase then issues a PAUSE to await completion of the Remote Phase.

The Remote Phase now proceeds as follows:

- The IRB routine passes control directly to Z/XDC proper so that it can produce its displays and act upon user commands.
- The user can use the **LIST LOCKS** command to see what locks were held (and now are lost).
- If the user wants to change the set of locks that will be reacquired upon program resumption (TRACE or GO/GOT/GOX command), he can use the **SET LOCKS** command to do that.
- If the user attempts to resume execution of the interrupted routine (TRACE or GO/GOT/GOX commands), and if locks had been held when Z/XDC received control, then a security call is made to see if the user has permission to resume his program under these circumstances. The security rule that is tested is named **XDC.LOSTLOCKS.** See HELP SECURITY LOSTLOCKS for more information.

WARNING! Unlike, all other security calls made by Z/XDC, for the XDC.LOSTLOCKS rule, if the security system returns a NOT PROTECTED response, then Z/XDC will **deny permission** for the TRACE or GO/GOT/GOX command to proceed. In other words, in order to continue with the debugging session, an XDC.LOSTLOCKS rule must exist, and it must give you READ access to that rule. Again, see HELP SECURITY LOSTLOCKS for more information.

- When Z/XDC proper completes (TRACE, GO/GOT/GOX or END command is successfully issued), Z/XDC returns to the IRB routine.

- The IRB routine then awakens the Local Phase (RELEASE is used), and then it (the Remote Phase) terminates.

The Local Phase now resumes and proceeds as follows:

- If TRACE or GO/GOT/GOX was issued, then the program's original environment is restored (including all locks originally held, but as modified by a possible SET LOCKS command), and then control is passed to the resume address.

- If **END** was issued, then the ABEND is **not** percolated. Instead, the program either is **reABENDed** or is terminated **"normally"** (i.e. ended but not ABENDed). The choice is made by an optional, user provided exit routine called an **FRREND** exit routine.

  If no **FRREND** exit routine is provided, then the default is to terminate the FRR protected routine **"normally".**

If you wish to provide an **FRREND** exit routine, you have to build a **DBCPARM** control block and provide the exit's address via one of its fields **(DPFRREND).** For more information, see HELP EXITS FRREND.

If the FRR protected routine is **reABENDed,** then that is accomplished by executing a **DEAD trap** containing a **DBC557T** message. See HELP MESSAGES DBC557 for more information, especially information about register settings at the time of ABEND.

If the FRR protected routine is terminated **"normally",** then that is accomplished as follows:

- If The FRR was protecting an **SRB** routine, then that routine is terminated via a jump to **CVTSRBRT.**
- If The FRR was protecting a task-mode **RB** routine, then then that routine is terminated via an **SVC 3** instruction.

Prior to terminating, a **return code** (of sorts) is loaded into **R15.** The code is **C'XDC '.** Note, this return code **does not change** to match Z/XDC's current clone name.

Cross Memory Mode

Z/XDC can be used to debug FRR protected routines that run in cross memory mode. It takes the steps necessary to preserve the program's cross memory environment during Z/XDC's processing.

About Lost Locks

As noted above, when Z/XDC receives control as an FRR routine, the nature of interactive debugging requires that it go into a wait state. (z/OS's PAUSE/RELEASE services are used.) This, coupled with the nature of Z/XDC's task mode processing, necessitates that if the user's routine held any locks, they must **all** be freed prior to control being passed to the Remote Phase of Z/XDC processing! This, of course, has **profound consequences,** and this is the major reason why, as noted above, FRR mode debugging is so dangerous!

Obviously, when a lock is lost and reacquired outside of a program's knowledge, important things can break. Badly! But the flip side of that is that things might not break at all. It might be the case that the interfering activity that the lock is guarding against simply does not occur.

This actually is the case far more often than not. And when this is the case, the value of being able to use an interactive debugger is enormous!

For more information regarding the issues raised by lost locks, please see HELP DEBUGGING LOSTLOCKS.

Help DEbugging Hooks

This topic has been moved to **HELP HOOKS.**

Help DEbugging Ispf

Z/XDC can be used either from TSO's READY prompt or from within ISPF to debug user programs. Either way, Z/XDC will display a fullscreen interface, and Z/XDC will provide such ISPF-like features as comprehensive session scrolling, session logging, and session profiling services. See **HELP FULLSCREEN** for more information.

Z/XDC's session scrolling, logging, and profiling support is **identical** between ISPF and non-ISPF environments. Z/XDC's fullscreen display support, however, is only nearly identical. There are some unavoidable differences. See **HELP DEBUGGING ISPF DISPLAYS** for detailed information.

Within ISPF, a Z/XDC debugging session can be started in at least two ways:

- **The hard way:** From ISPF's **ISPF Command Shell** panel (ISPF Option **=6):**
  - Issue appropriate TSO **ALLOCATE** commands to allocate needed **TASKLIBs, XDCCMDS** and other files.
  - Then type an **XDCCALL, XDCCALLA, XDCCMD,** or **XDCCMDA** command to start the debugging session.
  - The **XDCCALL** (et al.) command will run, and it will ATTACH the user program as its subtask.
  - Z/XDC itself will be an **ESTAI** type ABEND recovery routine for the user

program.

-See **HELP DEBUGGING TSO** for more information.

- **The easy way:** Z/XDC has a **Z/XDC Startup Panel** in ISPF for automating the invocation of **XDCCALL** and its aliases. For information about the panel, see **HELP DEBUGGING ISPF PANEL.**

  Usually, the Z/XDC panel is installed as a sub-panel of ISPF's **Primary Option Menu** and can be reached by a simple **=D** jump command. However, your Systems Programmer may have installed the Z/XDC panel elsewhere in ISPF, so you may have to ask him where to find it. (For information about how the panel is installed, see **HELP DEBUGGING ISPF PANEL INSTALLING.)**

  In any case, when you reach the Z/XDC panel, you can fill in its fields to indicate:
  - The **program** that you want to debug,
  - Whether it is a **normal** program or a **TSO command,**
  - What its **PARM** field string or **commands operand** string is,
  - Whether or not it is to execute **authorized,**
  - What its **task libraries** are,
  - And so forth.

  When you finish filling out the panel and then press **ENTER,** the panel will use **XDCCALL** or one of its aliases to start a debugging session.

The debugging environment created when using the Z/XDC Startup Panel is essentially identical to that which is created by using the **XDCCALL** (et al.) command directly. For complete information about that environment, see **HELP XDCCALL.**

In particular, note that XDCCALL establishes Z/XDC as the user program's ABEND error **recovery routine** by ATTACH'ing the user program as a subtask and naming Z/XDC as the **ESTAI** routine for that subtask. This makes Z/XDC an error recovery routine, not only for the user program, but also for any subtasks that it might create as well. (This is because z/OS automatically propagates ESTAI type ABEND protection to all descendants of a protected task.)

In other words, when XDCCALL (or any of its aliases or Z/XDC's Startup Panel) is used to start a debugging session, then any ABEND or any breakpoint that occurs either within the user program or within any of its subtasks can be intercepted by Z/XDC. However, any ABEND or breakpoint that occurs under any *other* task cannot be intercepted by Z/XDC. Example:

- Suppose:
  - That Z/XDC's Startup Panel (or XDCCALL directly) was used to start a debugging session against a user program named "MULTUSER".
  - That MULTUSER has been allowed to execute to the point where it has ATTACH'd three subtasks name "COMMTASK", "WORKTASK", and "USERQUE".
  - Finally, that USERQUE has attached two subtasks of its own named "QIN" and "QOUT".

- Then if Z/XDC's **LIST TASKS** command is used to display this multitasking structure, the resulting display might look something like this:

```
...
 8  . . ISPMAIN
 9  . . . ISPTASK (ISPLLIB)
10  . . . . XDCCALL
```

```
      11  . . . . . "MULTUSER"
      12 . . . . . . USERQUE
      13 . . . . . . . QOUT
      14 . . . . . . . QIN
      15 . . . . . . WORKTASK
      16 . . . . . . COMMTASK, CURRENT (ABEND 0C1)
      17  . . . ISPTASK (ISPLLIB)
      ...
```

Note the following:

- The **LIST TASKS** command hilights those tasks that are a part of the debugging session.

- **ISPMAIN** is ISPF's main task.

- If you have used ISPF's **SPLIT** command, then two **ISPTASK** tasks will exist, and they will be sisters to each other; otherwise, only one **ISPTASK** will exist.

- All **non-authorized** programs and commands that run "under ISPF" (such as XDCCALL) will run as subtasks of one or the other of the **ISPTASK** tasks, depending upon whether that program is running above or below ISPF's split line.

- So long as MULTUSER and its subtasks do not issue any ESTAE (etc.) macros, Z/XDC can intercept any ABEND or breakpoint that occurs in the MULTUSER task (#11) or any of its descendants (tasks #12-16). This is because MULTUSER is a subtask of XDCCALL.

- ABENDs or breakpoints occurring in any other task will not be intercepted by Z/XDC unless special action is taken by or on behalf of the program's suffering the ABENDs.

  This fact is especially relevant for "Dialog Function" programs invoked via ISPF's ISPEXEC or ISPLINK service. Such programs are **not** executed within the subtask structure below XDCCALL. Instead, they are executed either directly under an ISPTASK or as a direct subtask of an ISPTASK. Consequently, debugging user written ISPF Dialog Functions requires special steps.

Subtopics Index

For more information, select the following topics. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

PANEL - Using Z/XDC's Startup Panel (in ISPF) to start debugging sessions.

DISPLAYS - Differences in Z/XDC's fullscreen displays in ISPF vs. non-ISPF environments.

DIALOGS - Special requirements for debugging ISPF dialog functions.

AUTHORIZED - Special considerations when debugging authorized programs under ISPF.

XDCLBDEF - A clist for those who do not want to add Z/XDC libraries to ISPPLIB, ISPTLIB, ISPMLIB and SYSPROC concatenations.

Help DEbugging Ispf Panel

Z/XDC has a **Z/XDC Startup Panel** in ISPF for automating the process of starting a debugging session. The panel does this by setting up several file allocations and then starting the debugging session by invoking **XDCCALL** or one of its several aliases.

Usually, the Z/XDC panel is installed as a sub-panel of ISPF's **Primary Option Menu.** So, it can be reached by a simple **=D** jump command. However, your Systems Programmer may have installed the Z/XDC panel elsewhere in ISPF, so you may have to ask him where to find it. For more information, see **HELP DEBUGGING ISPF PANEL INSTALLING.**

Using the Startup Panel to Start a Debugging Session

Command line options **1** and **2** are for debugging programs (option 2) and TSO commands (option 1) that run within the TSO address space. (See below about option **3** for debugging batch programs.)

Most of the Panel's fields apply to options **1** and **2** but not to option 3. These fields allow you to specify:

- The **program** that you want to debug.
- Whether it is a **normal** program or a **TSO command.**
- What its **PARM** field string or **command operand** string is.
- Whether or not the PARM data is to be **upcased** before the target program sees it. (For more information, see **HELP XDCCALL.)**
- Whether or not the target program is to execute **authorized.**
- Whether **RENT** and **REFR** modules are to be loaded into **key 0** storage or into **user-key** (usually key 8) storage.
- For c/XDC, whether or not execution should be **automatically advanced** from a module's Entry Point, past all the prolog code, through to the machine code backing the first language statement **(AUTOSTEP'd).**
- What its **task libraries** are (up to six of them).
- Whether your program is to be **quick started,** or Z/XDC is to receive control first.
- The name of a **commands script** (if any) that you would like Z/XDC to run at the start of the debugging session.
- Etc.

When you finish filling out the panel, select option **1** or **2,** and press **ENTER.** The panel will then use **XDCCALL** or one of its aliases to start a debugging session. (See **HELP XDCCALL** for more information.)

Connecting to a Debugging Session Running in the Batch

When you select option **3,** the panel can instead be used to connect, via cdf/XDC, to a debugging session that has already been started in a background batch job or system task. See **HELP XDCSRVER CDF** for more information about cdf/XDC.

Selecting option **3** causes **all of the panel's fields described above to be ignored!**

The only field that is not ignored is the one listed below the **PARAMETER FOR OPTION 3** caption:

- **Connect to cdf/XDC Using your TSO userid?**
  This controls whether (NO) or not (YES) you will have to go through a logon panel before starting your debugging session.

Running XDC Clones

Note, when Z/XDC is running under a **clone name** ("xxx") other than "XDC", then the panel's fill-in field named **XDC's name ===>** must be changed to match Z/XDC's actual name.

When you do this, you will see the panel automatically adjust to invoke Z/XDC via the new name instead of via "XDC".

Further, the change will persist from session to session until you decide to change it again.

For more information about renaming Z/XDC, see **HELP XDCCLONES.**

DDNAMEs Generated by the Startup Panel

When you use option 1 or 2, depending upon how you fill in the panel's various fields, the following ddnames may be allocated:

- **//xxxQUICK:** Related to the **Quick Start?** field.
- **//xxxRENT8:** Related to the **Should RENT and REFR ...** field.
- **//xxxREFR8:** Related to the **Should RENT and REFR ...** field.
- **//xxxNSTEP:** Related to the **Allow AUTOSTEP?** field.
- **//xxxT0006:** Related to the **Tasklib DSNs** fields.
- **//xxxCMDS:** Related to the **Script DSN** field.

More Information

Usage of the Z/XDC Startup Panel is fully documented by an ISPF tutorial. To display the tutorial:

- First display the Z/XDC Startup Panel itself.
- Then type ISPF's **HELP** command and press ENTER (or just press **F1).**

The tutorial will start.

Subtopics Index

For more information, select the following topics. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

INSTALLING - Integrating Z/XDC's Startup Panel into ISPF.

Help DEbugging Ispf Panel Installing

Preface: About HLQs

In this topic, there are several library names that are presented as **hlq.XDCV31.XDCwhatever.** The **hlq** refers to the name's **High Level Qualifier.**

Every customer creates their Z/XDC Product Libraries using their own, in-house High Level Qualifier. So, you will have to ask your SysProg what the **hlq** is set to on your system.

For more information about High Level Qualifies, see **HELP SUPPORT HLQ.**

Installing Z/XDC's Startup Panel into ISPF

To integrate the Startup Panel into ISPF, you must modify an existing ISPF panel. In principle, that could be any panel, but we find that many of our customers choose to put the panel link into ISPF's **Primary Option Menu (ISR@PRIM).** So, this discussion will presume that to be the panel to modify.

Proceed as follows:

- First, chose a Panel Selection Character. We recommend **D.**
- Next, you will have to modify the ISR@PRIM's panel logic in two places:
  - The **)AREA** section contains the panel's text that will be displayed. Insert the following line into the text:
    D XDC Extended Debugging Controller
  - The **)PROC** section contains the panel's processing logic. Within it, there is a **&ZSEL = TRANS** statement that translates Panel Selection Characters into commands that invoke the panel. This is where you need to insert a line that translates the **D** Selection Character into the command that leads to the display of the **Z/XDC Startup Panel.** BUT... There are two ways to go here:
    - **D,'PANEL(XDCPANEL) NEWAPPL(XDC)'**
      or
    - **D,'CMD(EXEC''hlq.XDCV31.XDCCLIST(XDCLBDEF)'')NEWAPPL(XDC)'**

    So, what's the difference?

D,'PANEL(XDCPANEL) NEWAPPL(XDC)'

This runs the **Z/XDC Startup Panel** directly. But in order for it to work, you need to add all of the following Z/XDC Product Libraries to their corresponding ddnames in your TSO logon proc (or startup clist):

- Add **hlq.XDCV31.XDCPLIB** to **//ISPPLIB**
- Add **hlq.XDCV31.XDCTLIB** to **//ISPTLIB**
- Add **hlq.XDCV31.XDCMLIB** to **//ISPMLIB**
- Add **hlq.XDCV31.XDCCLIST** to **//SYSPROC**

D,'CMD(EXEC ''hlq.XDCV31.XDCCLIST(XDCLBDEF)'') NEWAPPL(XDC)'

This runs a clist to perform **ALL** the necessary allocations. If you choose this method, you will not have to make any changes to your TSO logon time allocations.

Help DEbugging Ispf DISplays

Z/XDC can use either of three system interfaces for generating fullscreen displays and sending them to your terminal. We refer to these interfaces as

- The **TPUT** interface
- The **ISPF** interface
- The **VTAM** interface

Z/XDC uses the VTAM interface when the user is debugging a background job via cdf/XDC **and** the user has connected to cdf/XDC via a **LOGON APPLID=XDC** command from an idle VTAM terminal.

In all other cases (including connecting to cdf/XDC from an active TSO session), Z/XDC will use either its TPUT interface or its ISPF interface. (See **HELP XDCSRVER CDF** for more information.)

When using the TPUT interface, Z/XDC uses fullscreen type TPUT and TGET macros to generate its displays and to receive responses. Z/XDC uses the TPUT interface whenever it cannot use the ISPF interface and, optionally, even when it can.

When using the ISPF interface, Z/XDC uses ISPF's Display Services to generate its displays and receive responses. Z/XDC can use the ISPF interface only when Z/XDC has been invoked from within ISPF to debug either a **non-authorized** program running in the TSO address space or **any** background program (regardless of authorization) connected to Z/XDC via cdf/XDC.

At all other times, Z/XDC must use its TPUT interface. In addition, if you are using Z/XDC to debug an ISPF Dialog Function program that is manipulating ISPF variables and pools, then I recommend that you not use the ISPF interface. Using the TPUT interface instead avoids any possibility of interference by Z/XDC with your program's ISPF variable pools.

Z/XDC reports in its screen's title line which display painting method it is using. Example: "XDC **TPUT** INTERFACE".

When the ISPF interface is available for use, you can switch back and forth between the ISPF interface and the TPUT interface instantly by using the **SET ISPF** and **SET TPUT** commands. Z/XDC will immediately begin using the specified interface if it can. If you issue the **SET ISPF** command when Z/XDC is unable to use the ISPF interface, then Z/XDC keeps the request pending but otherwise ignores the command.

If Z/XDC has been using the ISPF interface, and conditions arise whereby it can no longer use that interface, then Z/XDC will automatically switch to using its TPUT interface.

Your preferred interface choice can be saved in your session profile. Z/XDC's factory default is to use the ISPF interface when possible. However, you can change this if you want to. See **HELP PROFILES** for more information.

The TPUT and ISPF interfaces are nearly identical in appearance and usability. The only differences are the following:

- The screen's title line reports which interface is in use: TPUT or ISPF.
- When the TPUT interface is in use, ISPF dependent commands cannot be used. These commands are **SPLIT, SWAP** and **=whatever** [an ISPF =jump command].
- When the ISPF interface is in use, the ISPF dependent commands can be used.

Other "ISPF like" commands are actually implemented by code within Z/XDC itself, so they can be used regardless of the interface. These commands are:

- **KEYS**
- **TSO**
- **UP**
- **DOWN**

Still other ISPF commands are not supported at all by Z/XDC. Example: PFSHOW

Help DEbugging Ispf DIAlogs

Z/XDC can be used to debug ISPF Dialog Function programs written in Assembler. There are, however, some special considerations that need to be discussed.

A Dialog Function is a program that can be invoked via ISPF's **ISPEXEC** or **ISPLINK** service. and it may invoke other programs or ISPF services via the same interface. (See IBM's "ISPF Dialog Management Guide", SC34-4213, and "ISPF Dialog Management Services and Examples", SC34-4215, for more detailed information.)

When "program A" invokes "program B" via the **ISPEXEC** or **LINK** service, it does not matter in what task program A is executing, program B will always run either as a subroutine of or a direct subtask of one of the **ISPTASK** tasks. Consequently, if you

use **XDCCALL** or the **Z/XDC Startup Panel** to debug programs A and B, you will easily be able to debug program A, but unless you take further steps, you will have problems debugging program B. This is because program A will run as a subtask of **XDCCALL,** but program B will run as a direct subtask of an **ISPTASK,** and so it will not be a descendant of XDCCALL. Example:

```
 LIST TASKS
      ...
 8  . . ISPMAIN
 9  . . . ISPTASK (ISPLLIB)
10  . . . . PROGRAMB
11  . . . . XDCCALL
12  . . . . . "PROGRAMA", CURRENT (ABEND 0C1)
13  . . . ISPTASK (ISPLLIB)
      ...
```

In order to debug program B, you must take the steps necessary to make Z/XDC the newest ABEND error recovery for program B (just like it is for program A). In general, this can be done in any of several ways:

- If you can change the source code for program B, then code an ESTAE macro in program B that makes Z/XDC its newest ABEND error recovery routine. a Z/XDC interface can be created in any program, including Dialog Function programs, just by LOAD'ing the Z/XDC module into storage, and then issuing an ESTAE macro that points to Z/XDC as the ESTAE routine. Example:

  ```
  XDCSETUP LOAD  EPLOC==CL8'XDC',ERRET=NOXDC
           ESTAE (0)
  ```

  ("==" is **not** a typo.) For more detailed information, see **HELP DEBUGGING EXITROUTINES.** Even though that discussion concerns itself primarily with coding Z/XDC interfaces into various kinds of system, product, and programming exits, the information presented is applicable to this situation as well.

- Alternatively, instead of a LOAD/ESTAE sequence, you can code a **#XDCHOOK** macro to assemble a static hook into your code. That will create a debugging session when your program's execution reaches it. See **HELP HOOKS STATIC #XDCHOOK** for more information.

- If you don't want to modify program B's source code, then you can use Z/XDC's **HOOK** or **HDEFERRED** command to set a hook dynamically into program B code at execution time.

  - Use the **HOOK** command if program B has already been loaded into storage by the time you're ready to set the trap.

  - Use the **HDEFERRED** command if program B has not yet been loaded into storage.

Help DEbugging Ispf Authorized

IBM has designed ISPF to run as a non-authorized program that presents a non-authorized environment to the Dialog Functions and other programs that run under it. In fact, ISPF does not even permit authorized programs to use its services! (Which is why Z/XDC's **SET ISPF** command does not work when it is issued from an authorized debugging session running "under" ISPF.)

Even though ISPF rather aggressively defends its non-authorized status and environment, it nonetheless does provide a way for authorized programs to be started without requiring that ISPF be terminated first. Normally, if you run a non-authorized program, ISPF will ATTACH that program as a subtask and let it run as one of its descendant tasks. However, if you run an authorized program, ISPF will first check a special list (SYS1.PARMLIB(IKJTSOxx)) to see if that program is permitted. If so, then ISPF will call a special TSO service SVC that will cause the program to be ATTACH'd as a subtask of a TSO task, thus causing the program to be executed entirely **outside** of ISPF's branch of the address space's subtask tree. This makes it not possible for the authorized program to use ISPF services, since all ISPF services check to ensure that their callers are descendant tasks of ISPF.

Consider these examples.

- First, suppose you jump to ISPF's **ISPF Command Shell** panel and type **XDCCALL IEFBR14** to start a non-authorized debugging session. Then suppose that you issued Z/XDC's **LIST TASKS** command to display the address space's subtask structure. The result would look something line this:

```
list tasks
TCB# PROGRAM NAME (ASID 007D, DBCOLE6)
1    IEAVAR00
2    . IEESB605
3    . . IKJEFT01, JOBSTEP (STEPLIB)
4    . . . IKJEFT02
5    . . . . IKJEFT09
6    . . . . . ISPF
7    . . . . . . ISPTASK (ISPLLIB)
8    . . . . . . ISPTASK (ISPLLIB)
9    . . . . . . . XDCCALL
10   . . . . . . . . "IEFBR14", CURRENT (ABEND: S0C1)
11   . IEAVTSDT
```

  The IEFBR14 task, being non-authorized:
  - Is allowed to run as a descendant of the ISPF task.
  - Therefore, it can use ISPF services.
  - Therefore, Z/XDC, which is running within the IEFBR14 task, can use ISPF services.

  - Therefore, Z/XDC's **SET ISPF** command can be used to enable its ISPF interface.

  - Therefore, ISPF's **SPLIT** and **SWAP** commands can be used to switch from Z/XDC to another program in ISPF.

- Now suppose that you instead type **XDCCALLA IEFBR14** to start an authorized debugging session. Then if you issue a **LIST TASKS** command, the result would now look something like this:

```
list tasks
TCB# PROGRAM NAME (ASID 007D, DBCOLE6)
1    IEAVAR00
2    . IEESB605
3    . . IKJEFT01, JOBSTEP (STEPLIB)
4    . . . IKJEFT02
5    . . . . XDCCALLA
6    . . . . . "IEFBR14", CURRENT (ABEND: S0C1)
7    . . . IKJEFT02
```

```
        8    . . . . IKJEFT09
        9    . . . . . ISPF
        10   . . . . . . ISPTASK (ISPLLIB)
        11   . . . . . . ISPTASK (ISPLLIB)
        12   . IEAVTSDT
```

The XDCCALLA task, being authorized:
- Is **not** allowed to run as a descendant of the ISPF task.
- Instead, a second **IKJEFT02** task (known as a "sister TMP") is created by ISPF,
- And XDCCALLA is forced to run there, entirely outside of ISPF.
- Therefore, neither XDCCALLA nor its descendants (IEFBR14) can use ISPF services.
- Therefore, the **SET ISPF** command cannot be used to enable Z/XDC's ISPF interface.
- Therefore, ISPF's **SPLIT** and **SWAP** commands **cannot** be used to switch from Z/XDC to another program in ISPF.

Help DEbugging Ispf Xdclbdef

What It Is

XDCLBDEF is a clist that a customer can use to run the **Z/XDC Startup Panel** without having to actually install the panel into ISPF.

XDCLBDEF uses ISPF's **LIBDEF** service and TSO's **ALTLIB** command to temporarily set up:
- A panel library (ISPPLIB)
- A table library (ISPTLIB)
- A messages library (ISPMLIB)
- And a clist library (SYSPROC)

It then uses ISPF's **SELECT** service to run the **Startup Panel.**

Where We Put It

XDCLBDEF is available in the Product Library named **hlq.XDCV31.XDCCLIST** (where **hlq** is the library name's High Level Qualifier).
- Every customer creates their Z/XDC Product Libraries using their own, in-house High Level Qualifier.
- You will have to ask your SysProg what the hlq is set to at your site.
- For more information about High Level Qualifies, see **HELP SUPPORT HLQ.**

Where You Can Put It

Your installer can either leave XDCLBDEF in the hlq.XDCV31.XDCCLIST library, in which case you will have to use TSO's **EXEC** command to run it from there.

Example: **EXEC 'hlq.XDCV31.XDCCLIST(XDCLBDEF)' 'operands'**

Or your installer can copy it into one of your local clist libraries (VENDOR.CLIST, perhaps), and then you can just run it by name. Examples:
- **%XDCLBDEF operands**
- **TSO %XDCLBDEF operands**

Ask your SysProg which is the case.

For the rest of this topic, we will presume that XDCLBDEF has been copied into one of your local clist libraries, since that greatly simplifies the command needed to run the clist.

How to Run It

XDCLBDEF can only be run from withing ISPF. It cannot be run from TSO's READY prompt.

Within ISPF, it can be run:
- Either directly from ISPF's Command Shell **(=6),**
- Or indirectly from anywhere via ISPF's **TSO** command.

When XDCLBDEF has been copied into one of your local clist libraries, it can be invoked by name. Use of TSO's **EXEC** command is not necessary.

Syntax

```
    [TSO] %XDCLBDEF PLIB(libraryname)  +
                    TLIB(libraryname)  +
                    MLIB(libraryname)  +
                    CLIST(libraryname) +
                    CLONE(xxx)         +
                    LIST               +
                    DEBUG
```

Notes:

- All operands are optional.

- For the several **libraryname** operands, the defaults are various Z/XDC's Permanent Product Libraries.

- For the **CLONE(xxx)** operand, the default is **XDC.**

Operands

PLIB(libraryname)
TLIB(libraryname)
MLIB(libraryname)
CLIST(libraryname)

These operands identify the names of the libraries that contain the various elements of Z/XDC's interface for ISPF.

The factory defaults are the names of libraries that are among those that comprise your site's Primary Product Libraries, as created by the SysProg that applies product maintenance. Those libraries are:

- hlq.XDCV31.XDCPLIB
- hlq.XDCV31.XDCTLIB
- hlq.XDCV31.XDCMLIB
- hlq.XDCV31.XDCCLIST

The High Level Qualifier is particular for your site. Please ask your SysProg what it is. For more information, see **HELP SUPPORT HLQ.**

CLONE(xxx)
Most customers don't do this, but just in case...

If you are running a copy of XDC that has been renamed from "XDC" to some other 3-character name, then this operand tells the clist what that name is. For more information about clones, see **HELP XDCCLONES.**

LIST
This causes the clist to issue TSO's **CONTROL LIST** command, which causes linemode messages to be displayed showing the various ISPF and TSO commands that the clist issues under the covers.

DEBUG
This causes the clist to issue TSO's **CONTROL LIST,SYMLIST,CONLIST** command, which causes linemode messages to be displayed showing the clist's executed logic in all its gory detail.

Help DEbugging Jes2

Z/XDC is a very powerful tool for developing and debugging both exits and source code modifications to JES2. Z/XDC can provide a fullscreen debugging interface for both secondary and primary JES2's. Breakpoints can be placed anywhere in main task code, in exit routines, and (with some additional effort) in JES2 subtask code.

ColeSoft provides a pre-written Z/XDC interface for JES2 that allows you to use Z/XDC to debug source code mods and JES2 exits that run in JES2's main task. This interface consists of four independent components:

- A type-5 exit that implements a **$XDC ON/OFF** command that allows you to turn Z/XDC debugging on or off within JES2.

- A type-0 exit that implements an **XDC** startup parameter that allows you to use

Z/XDC to debug JES2 initialization code.

- Source code mods to JES2's **$CB** and **$MODULE** macros that should be used if you are making other source code mods to JES2 and you want to use csect maps in your work.

- A reassembly with PARM=TEST of the HASPDOC source module to make JES2 dsect maps available for use by Z/XDC.

Each of these four components is entirely independent of the other three. Each can be installed or left out in any combination with the other three. For complete information about installing the Z/XDC interface into JES2, please see Z/XDC's **Installation Guide.**

If you need to debug exits that run within subtasks of JES2 (such as the type-6 exit that runs in the Converter/Interpreter subtask), you will have to code a Z/XDC interface directly into the exit itself. The prewritten Z/XDC interface does not directly support debugging subtask code in JES2.

One very powerful and very easy-to-use way to create a Z/XDC debugging session within a JES2 subtask is to use Z/XDC's **HOOK** command. See **HELP HOOKS** for more information.

In order to run a fullscreen debugging session against JES2 (or JESx), you should change the JES2 (or JESx) startup proc as follows:

- JES2 debugging sessions **cannot** be started within an XDCCALL environment; therefore, the EXEC card's **PGM=HASJES20** parameter should **not** be changed.

- If you want to use a Z/XDC that has been renamed from **XDC** to some other name (such as **V31),** then you will have to add the following DD card to notify Z/XDC's JES2 exits what Z/XDC's name actually is:

  //XDCISxxx DD DUMMY

  where **xxx** matches Z/XDC's name. For more information, see **HELP XDCCLONES.**

- Normally, when Z/XDC receives control it will attempt to use cdf/XDC for communicating with the programmer. (See **HELP XDCSRVER CDF.)** However, cdf/XDC requires that VTAM be up and running. VTAM, however, doesn't start until after the primary JES2 is started. So when debugging the primary JES2, it can occur that you may want to debug JES2 code that runs prior to the starting of VTAM. If this is the case, then add the following DD card to the JES2 startup proc:

  //xxxWTOR DD DUMMY

  (where **xxx** matches Z/XDC's name). The presence of this DD card causes Z/XDC to use WTO/WTORs (instead of cdf/XDC) to communicate with the programmer via a Systems Operator's console. (This problem generally does not occur when debugging a secondary JES2.)

- If you want to use initial profile settings that are different from Z/XDC's factory defaults, then add an //XDCPROF DD card that points to a profile library (RECFM=FB, LRECL=80, BLKSIZE=n*80). Example:

  //xxxPROF DD DISP=SHR,DSN=SYS1.JES2.PROFILE

  where **xxx** matches Z/XDC's name. Warning: the dataset must be cataloged in your system's Master Catalog. Also, this library can be a copy of your ISPF profile library, but don't use the same library; otherwise, you will run into sharing conflicts with your TSO session.

- Include whatever other DD cards the JES2/JESx startup proc needs.

Example:

```
//JES2    EXEC PGM=HASJES20,TIME=1440,...
//XDCWTOR  DD  DUMMY
//XDCPROF  DD  DISP=SHR,DSN=SYS1.JES2.PROFILE
//  <other DD cards needed by JES2>
```

When debugging secondary JES2s, you will be able to use cdf/XDC (and, therefore, be able to debug from a fullscreen terminal) throughout the session no matter where in JES2's code to want to debug. For the primary JES2, however, cdf/XDC cannot be used prior to the completion of JES2's initialization and the issuance of the initial **$S** command. This is because cdf/XDC requires VTAM, and VTAM cannot start prior to the initial **$S.**

When cdf/XDC cannot be used, you can add the **//xxxWTOR DD DUMMY** JCL card to cause Z/XDC to use its WTO/WTOR interface for communicating with the programmer. This interface is "linemode" by nature but nonetheless provides access to Z/XDC's very powerful debugging capabilities. See **HELP USERINTERFACES** for more information.

One very powerful tool that you will need access to for debugging JES2 is a set of dsect maps for JES2's control blocks. You can create these maps very easily. Just assemble the IBM provided HASPDOC module specifying PARM='TEST' for the assembly. Then linkedit the result also specifying **PARM='TEST'** for the linkedit. Place the load module into a linklist library. You now have a repository of dsect maps for nearly all of JES2's control blocks. These maps can be read with Z/XDC's **DMAP** command, and they can be assigned to appropriate storage locations with the **USING** command. See **HELP MAPS** for more information.

Help DEbugging LE

IBM's "Language Environment" establishes a consistent subroutine, error handling, and interfacing environment for programs written in a number of different programming languages, including Assembler. Z/XDC can be used to debug Assembler programs executing within an L.E. environment, but the user will have to establish certain L.E. run-time options before doing so.

Whenever Z/XDC receives control within an L.E. environment, it makes a call to an L.E. handshaking module named CEE3ERP. This module gives L.E. the opportunity to process those ABENDs and program checks that it needs to process and to let Z/XDC deal with the rest. If you want to find out more about CEE3ERP, please read IBM's **Language Environment Vendor Interfaces.** You can find it online by Googling the title.

In order for Z/XDC to be used within an L.E. environment, whenever a breakpoint, program check, or ABEND occurs, Z/XDC must receive control first, ahead of L.E.'s recovery routines. The easiest way to accomplish this is by using the:

HOOK command,
HDEFERRED command,
or #XDCHOOK macro

to place a hook into your Assembler code at the point where you want to start debugging. The hook, when executed by your code, will create a Z/XDC environment and then use that environment to pass control to Z/XDC. See HELP HOOKS for more information.

The Z/XDC environment that the hook creates will supersede L.E.'s own error recovery environment, but then when Z/XDC calls CEE3ERP, L.E. will at that time be made aware of the ABEND or program check, and so it will still have the opportunity that it needs to perform whatever action it needs to take.

In broad terms, as far as L.E. is concerned, ABENDs and program checks fall into three categories:

- Those that L.E. itself expects and can handle. (I'll refer to these below as "category A" errors.)
- Those that L.E. does not expect, but that the user program does expect and can handle, i.e. those for which the user program has established "handlers" ("category B" errors).
- Those that no one expects and that no one can handle. ("category C" errors).

Language Environment's error recovery environment normally consists of an ESTAE routine and an ESPIE routine. When using Z/XDC, it is very important that L.E.'s ESPIE routine not exist. (See HELP DEBUGGING ESTAES ESPIES for the reasons why.) Fortunately, L.E. provides a very easy way for the user to cause L.E. to suppress its ESPIE routine: The **TRAP** run-time option.

TRAP(ON|OFF,NOSPIE)
This causes L.E. to handle all expected error conditions (category A and B errors), but to do so without the use of an ESPIE. (L.E. will handle the program checks via its ESTAE when it sees the 0Cx ABEND caused by the program check.) When Z/XDC is setup to receive control ahead of L.E.'s ESTAE, L.E. will still be given the opportunity to process the errors when Z/XDC calls the CEE3ERP handshaking routine. Z/XDC will continue to process the ABEND only if CEE3ERP reports that L.E. has not processed the ABEND (category C errors).

TRAP(ON|OFF)
This causes L.E. to handle only category A errors. Category B and C errors will be processed by Z/XDC.

There are two ways to pass run-time options to LE:

- Put them into a sequential file, and use ddname CEEOPTS to point to it. Example:

```
// EXEC PGM=XDCCALL,
//      PARM='mypgm myparms'
//CEEOPTS  DD  *
leruntimeoptions
/*
```

- Put them into PARM field. Use a slash (/) to separate them from your program's own parameters. Example:

```
// EXEC PGM=XDCCALL,
//      PARM='mypgm leruntimeoptions/myparms
        ***or***
//      PARM='mypgm myparms/leruntimeoptions
```

(Apparently, some LE environments expect the leruntimeoptions to come ahead of your program's PARMs, while other LE environments expect them to come after your PARMs. Go figure!)

For more information about the TRAP run-time option and about run-time options in general, please see the following IBM manuals:
- z/OS Language Environment Programming Guide
- z/OS Language Environment Programming Reference

Help DEbugging LOstlocks

One of Z/XDC's very powerful capabilities is the ability to debug code that holds locks! But of course there is a catch. In order to debug such code interactively, the locks have to be released before Z/XDC can do anything useful... Of course, this sounds horrific, but let's look at the issues closely and see if it really is as bad as all that.

About Lost Locks

When Z/XDC receives control, either as an FRR or as an ESTAE-type routine, the nature of interactive debugging requires that if the user's routine held any locks, they must **all** be freed prior to Z/XDC beginning its normal processing. This, of course, has **profound consequences!**

When Z/XDC is running as an FRR, if the program being debugged is resumed (via TRACE or GO), then the released locks are reacquired prior to control being returned to the user program. This is **not** done for the purpose of maintaining assured integrity. (Assured integrity was lost, of course, the moment the locks were released.) But restoring the locks remains beneficial because the program being debugged expects the locks to be set and will fail in additional miserable ways if they are not.

Unfortunately, when Z/XDC is running as an ESTAE-type routine, it cannot automatically restore lost locks. That's because for ESTAEs, z/OS releases the locks before passing control to ESTAEs, and although it will report which locks were released, it does not preserve any lock attribute information necessary for restoring many locks.

Example: When a Local Lock is released by RTM (prior to giving control to ESTAEs), RTM will note in the SDWA that a local lock was releases, but it will not record the ASID of the Local Lock. So there is no way to know if it was the local Local Lock or a Cross Memory Local Lock of some other aspace! [sigh]

The Issues and Risks Raised by Lost Locks

Obviously, when a lock is lost and reacquired outside of a program's knowledge, important things can break. Badly! But the flip side of that problem is that things might not break at all. It might be the case that the interfering activity that the lock is guarding against simply does not occur.

This actually is the case far more often than not. And when this is the case, the value of being able to use an interactive debugger is enormous!

So the important questions are:

1) What are the risks of losing your locks?
2) Are the benefits of the debugging session worth the risks?

The risks are dependent upon many factors. To evaluate them, the following questions need to be asked.

- What locks are involved? Are they critical to your program? How critical are they to the System?

- What kind of damage could result in the event that the loss of a lock results in corruption?
  - Could the resulting damage confuse the purpose of the debugging session?
  - Could it bring down your SRB or task?
  - Could it bring down the address space?
  - Could it bring down the entire system?

- How likely would it be for a corrupting event to occur?
  - Is the event being delayed a frequent event?
  - Is it a rare event?

  Clearly, some potential events are going to be so frequent that their occurrence will be a certainty. But on the other hand, some events will be so rare, that (for debugging purposes) you can simply disregard the potential.

- Is the occurrence of a corrupting event controllable by you? Can you take steps externally to reduce the likelihood or even prevent a corrupting event from occurring?

- How serious would system corruption be?
  - Are you on a sandbox system where the only casualty will be yourself?
  - Are you on a development system where the only casualties will be your friends? Will they remain your friends after several crashes? (How many crashes?)
  - **Are you on a production system where a crash could lead to jail time?**

All of these concerns (and probably more) have to be considered whenever you think about debugging lock protected code. When feasible, the interactive debugging of such routines can be a boon to your productivity. On the other hand, when not feasible, the results can range from confusing to criminal negligence! **Be careful**!

The XDC.LOSTLOCKS Security Rule

In order to give your Data Center’s management a say in this decision process, Z/XDC supports a security rule to allow management to control where this risky debugging may take place, and who may engage in it. The guidelines are simple:

- This debugging should be permitted only on systems where the consequences of data corruption and frequent crashes are acceptable.

- Never define a PERMIT to this rule for any but those programmers who are the most experienced both with writing FRR protected code and with the meanings and uses of System Locks, as well as the detailed consequences of losing them.

As a further measure of protection, for the XDC.LOSTLOCKS rule, if the security

system returns a NOT PROTECTED response, then Z/XDC will **deny permission** for the TRACE or GO command to proceed. In other words, in order to continue with the debugging session, **an XDC.LOSTLOCKS rule must exist,** and it must give you READ access to that rule. (Note, this is unlike all other security calls made by Z/XDC, where a NOT PROTECTED response is treated as a PERMIT.)

See HELP SECURITY LOSTLOCKS for more information.

Help DEbugging PCroutines

Z/XDC can be used to debug any kind of PC routine:
- Space switching and nonswitching,
- Stacking and nonstacking,
- Supervisor state and problem state,
- System key and user key,
- Whatever.

In all cases, the PC routine must establish Z/XDC as its newest recovery routine:
- Either as an Associated Recovery Routine (ARR - not recommended),
- Or through the ESTAEX macro service (preferred over ARR),
- Or by setting a hook with Z/XDC's **HOOK** command. (This is the easiest).

Space Switching

When a program executes a PC instruction, execution may transfer to a PC routine located in an address space that is different from the address space in which the PC instruction is located. When this transfer occurs, the user program's Primary Address Space switches from being the space in which the PC instruction is located (often the home space) to being the address space within which the PC routine resides.

When debugging Space Switching PC routines, the XDC and XDC31 load modules must reside in common storage. This is because internally Z/XDC has to do its own space switching in order to function properly. But this space switching is entirely internal to Z/XDC and is managed so as to have no affect upon your program and its PC routine.

The internal space switching done by Z/XDC during its processing sets the primary aspace equal to the secondary aspace equal to the home aspace (i.e., P=S=H) so that Z/XDC may issue SVCs during its processing. These changes are undone before the PC routine being debugged executes its next instruction.

If the PC routine is executing in supervisor state, the changes are done using the LASP instruction and the changes can always be done.

If the PC routine is executing in problem state, the changes are done using the PT (or PTI) instruction, and the current primary aspace's Authorization Index (AX) must allow the setting of the home aspace as the primary aspace. If this cannot be done, an ABEND S0D7-25 will occur, and the problem state PC routine cannot be debugged by Z/XDC

Problem State PC Routines

As mentioned in the **Space Switching** section, there are some cases where a problem state PC routine **cannot** be debugged by Z/XDC.

Additionally, Z/XDC **cannot** be used to debug **space switched** PC routines that are running:
- In problem state,
- **AND** with a user execution key (key 8 thru 15),
- **AND** with a user key-mask (X'00xx'),

all at the same time. **But** if any one of these attributes is changed, then Z/XDC can be used just fine.

For example: in order to use Z/XDC to debug a problem state and user key, space switching PC routine, all you have to do is change the key-mask to permit execution key 0 (X'80xx' for example).

ESTAEX vs. ARR vs. HOOK Command

For PC routines, Z/XDC can be established either as an ESTAEX or as an ARR or via a HOOK command. When established as an ARR, Z/XDC provides debugging services for **all** callers of the PC routine from any address space in the System. This probably is not desirable.

To avoid this problem, consider establishing Z/XDC as an ESTAEX, but precede your ESTAEX macro call with filtering logic that issues the ESTAEX only when the calling address space is suitable. For an approximate example of such logic, see HELP DEBUGGING EXITROUTINES EXAMPLES.

Normally, to setup Z/XDC as an ARR or ESTAEX for a PC routine, you need to change the source code, either of the PC routine (for ESTAEX) or the PC routine's setup routine (for ARR).

However, if you want to avoid making a source code change, you can, instead, use the **HOOK** command to create dynamically the necessary ESTAEX (or FRR). For more information, see HELP COMMANDS HOOK.

If you want to set up Z/XDC as an ESTAEX, then your PC routine will have to contain logic that finds an in-storage copy of the XDC load module and then issues an ESTAEX macro establishing the XDC load module as your PC's ESTAEX type recovery routine. Unfortunately, it is not legal to issue a LOAD from within a PC routine; therefore, you will have to provide some other method for finding the XDC load module. See HELP DEBUGGING EXITROUTINES for suggestions. **(Or just use the HOOK command instead.)**

If you prefer to run Z/XDC as an ARR, then consider not pointing it to Z/XDC directly. Instead, write a simple front-end routine for Z/XDC and put your filtering logic there. The front-end would call Z/XDC only when the calling address space is suitable.

Note, Z/XDC's linkage conventions are the same as for ARRs.

For information about creating PC routines and establishing ARRs, start with IBM's "MVS Programming: Extended Addressability Guide" (GC28-1769).

Security

Debugging a space-switching PC routine is subject to the XDC.FASM.--- resource access control rules (RACF). The caller of the PC routine (as opposed to the PC routine's owner) must be permitted by the security facility to have WRITE access to the XDC.FASM.--- resource that controls Z/XDC's access to the address space that owns the PC routine. See HELP SECURITY FASM for more information.

Help DEbugging PLpa

When Z/XDC is executed within an authorized environment, the user has (subject to restrictions by Security System rules) the ability to use Z/XDC to zap and to set breakpoints into nearly all areas of common storage **including** "Store Protected" areas of common storage. This means that Z/XDC can be used to zap:

- Anything in the PLPA.
- Anything in the System Nucleus including the "read-only" areas.
- Anything in the MLPAs and FLPAs, regardless of whether or not they are defined as "read-only".

This means, in particular, that Z/XDC can be used to debug programs located in the PLPA. HOWEVER, **GREAT CARE MUST BE TAKEN** TO AVOID CRASHING OTHER ADDRESS SPACES AND TO AVOID CRASHING THE ENTIRE SYSTEM! In fact, using Z/XDC to debug programs located in common storage should not be attempted by anyone not having a thorough understanding of reentrancy, of z/OS in general, and of the environments within which his target program executes. In any case this should never be attempted on production systems, and a properly secured production system will not permit users to make such attempts. (See HELP SECURITY.)

Nonetheless, in a suitable test environment, Z/XDC's ability to zap and to breakpoint PLPA routines can be a very powerful tool for debugging system exit routines and other programs that might be located there.

In a likely scenario, you might need to debug an SMF exit, for example, located in the PLPA. Z/XDC can be used to do this, **but** if you want to avoid crashing other address spaces, you **must** be aware of and understand the following:

- SMF exits (for example) are executed frequently and even simultaneously by **every** address space in the System! However, you will want only one of those address spaces (your own) to execute the Z/XDC interface setup code so that extraneous Z/XDC debugging sessions are not started in address spaces other than the one with which you are working. Therefore, your Z/XDC interface setup code must include a test for the identity of the address space, and it must bypass starting a debugging session for all address spaces except the "right" address space. See HELP DEBUGGING EXITROUTINES EXAMPLES for an example of such a test.

- Anything that you zap in your exit routine (as well as anywhere else in common storage) is instantly visible to all address spaces in the System. Therefore, any zaps that you make must be either correct or benign if you want to avoid crashes.

- A corollary to the above is that you can make zaps that affect your debugging session from any authorized (and permitted) Z/XDC debugging session running in

any address space. This means that if the address space from which you are running the debugging session seizes up, you can still use an authorized Z/XDC running in another address space to make relevant zaps and possibly to correct the reason for the seizure.

- Z/XDC breakpoints, as far as z/OS is concerned, are nothing more than s0C1 ABENDs. Only Z/XDC understands when an s0C1 is actually a breakpoint. If you use Z/XDC to place a breakpoint into common storage code, then if that breakpoint is executed by any other address space, that address space will fail with an s0C1! In the case of high usage routines such as SMF exits, a carelessly placed breakpoint will create s0C1 failures faster that you can imagine. It is a quick and easy way to turn a healthy operating system into a smoldering heap of ashes. Thus, great care must be taken to place breakpoints in such a way that other address spaces do not try to execute them.

There basically are two ways to place breakpoints into common storage code such that they are unlikely to be executed by a wrong address space:

- You can place breakpoints into code that you "know" will not be executed by other address spaces. (Of course, you had better be right!) This selectivity might be either natural or artificial:
  - "Natural" selectivity occurs when the characteristics of the debugging session's environment are unique in a way that the code being tested selects for.
  - "Artificial" selectivity can be achieved by temporarily adding to your source code gates that test for your particular address space and cause all other address spaces to bypass the area in which you want to work.

- If you use a breakpoint that has only a "brief" existence, then execution of that breakpoint by other address spaces can become extremely unlikely. Note that breakpoints set by Z/XDC's TRACE and TRAP commands are "transient". They are automatically removed by Z/XDC when they are reached by the debugging session's execution. Breakpoints set by the AT command, on the other hand, must be removed manually by an OFF command. Accordingly, please observe the following:
  - By "brief", I mean existences that are measured in terms of microseconds or milliseconds at most. If the life of a breakpoint is long enough to be perceptible to humans, then its life is too long. It is not "brief".

  - Therefore, you should not set breakpoints in PLPA routines using the AT command or the ATX command, since the minimum life of an AT-type breakpoint includes at least the time necessary for you to type an OFF command, and that is not brief.

  - If you set a breakpoint using a TRAP command, then always follow the TRAP command with a ";GOT" command typed as part of the **same** command string. Example: "TRAP .+38 .TOPLOOP;GOT". Do not type the TRAP command and the GO command as separate command strings, because if you do, then the life of the trapping breakpoint will include at least the time necessary for you to type GO, and that is not brief.
  - When you set a breakpoint, you must be sure that it will be reached "quickly" by your debugging session's execution. Otherwise, its existence will not be brief, and the likelihood that it will be executed by another address space could reach dangerous levels.

- When execution reaches a breakpoint, it is important that Z/XDC remove **all** breakpoints from the code before issuing its displays and awaiting commands. Z/XDC will do this automatically if all existing breakpoints are transient (i.e. created by TRACE or TRAP commands) and if they all are assigned to the same breakpoint family. Therefore:

  - Do not issue two or more TRAP commands before issuing a GO command, because this will cause the generated breakpoints to be assigned to different families, and so they will not all be automatically removed when any one of them is reached by execution.

  - You may use one TRAP command to set two or more breakpoints, but be sure that at least one of those breakpoints will be "quickly" reached by execution.

  - Generally, you can freely use the TRACE command to step through execution. This is because the existence of TRACE type breakpoints is almost always brief.

As you can see, debugging active routines located in common storage can be very tricky and very prone to creating system crashes and address space ABENDs. It should be attempted by only the most experienced programmers.

Help DEbugging REEntrant

If you are running Z/XDC authorized, then you can skip this topic. This topic only matters when Z/XDC is running non-authorized.

The Problem

1) When load modules and program objects reside in key 0 storage and Z/XDC is running **non-authorized,** debugging is highly crippled.

2) Left to its own devices, z/OS will normally load **reentrant** modules into key 0 storage, thus severely limiting Z/XDC’s usefulness when debugging non-authorized programs.

3) Fortunately, there is a System supported method for causing reentrant programs to be loaded into key 8 storage regardless of their reentrancy attribute. Read on.

The Details

Generally, whenever z/OS initiates a program (authorized or non-authorized), it assigns that program to use **execution** key 8, and it sets the default storage key for its **GETMAINs** to be storage key 8 and fetch protected. Thus, as one would expect, a program is allowed to fetch from and store into storage that it owns.

In addition, **unless** a program is marked by the Linkage Editor (or Binder) as being reentrant (PARM=RENT) or refreshable (PARM=REFR), the **program’s own storage** also is

obtained from fetch protected key 8. This means that the program can both read from and write into its own code and assembled data areas.

However, if a program is marked **reentrant** (and **not** refreshable), then when it is loaded into the storage, z/OS may load it into fetch protected key 8 storage, **or** it may load the program into fetchable key 0 storage! If the **latter,** then the program can read its own code (because its storage is fetchable), but it cannot overwrite itself (because the storage key is 0 while the execution key is 8).

(Note, when a program is marked either as **REFR** or as **both** RENT and REFR, the refreshable attribute governs, and different considerations arise. See HELP DEBUGGING REFRESHABLE for details.)

Normally, the fact that the load module has been loaded into fetchable key 0 storage does not present any execution problem because by definition reentrant programs are not supposed to store into themselves anyway. **But** it does, under certain circumstances, make it difficult to use Z/XDC's breakpointing and zapping commands.

When a program is marked reentrant (and not refreshable), z/OS loads the program into fetch protected key 8 storage **or** fetchable key 0 storage depending upon from what library it is being loaded:

- If the reentrant program is loaded from an authorized library, then z/OS loads it into **fetchable** key 0 storage.

- On the other hand, if the reentrant program is loaded from a **non-authorized** library, then z/OS loads it into fetch **protected** key 8 storage.

- If the reentrant program is loaded from a concatenation that includes a mixture of authorized and non-authorized library, then z/OS treats the entire concatenation as being non-authorized and so it loads the program into fetch protected key 8 storage.

Notice that it doesn't matter whether or not the reentrant program itself is authorized. What matters is whether or not the program is loaded from an authorized library.

Possible Solutions

When Z/XDC is invoked non-authorized, you will not be able to set breakpoints (or zaps) into reentrant programs that are loaded into key 0 storage. Both breakpoint and zap attempts will fail with DBC045E STORAGE ACCESS VIOLATION - [...].

There are three ways to solve this problem:

- 1) **Poisoning the Concatenation:** When you want to debug a non-authorized, reentrant program, loaded from an authorized library, all you have to do is **fool z/OS** into thinking that the program is being loaded from a non-authorized library. To do this, simply allocate your program's load library as a STEPLIB, a JOBLIB, an ISPLLIB or a task library (whichever is appropriate) **concatenated** with any, arbitrary non-authorized library. z/OS will then treat the **entire concatenation** as being non-authorized, and so all **non-REFR** programs loaded from the concatenation will be loaded into fetch protected key 8 storage.

[Note, for dealing with modules that have been marked with **both** RENT and REFR, see HELP DEBUGGING REFRESHABLE.]

When your debugging session is started via XDCCALL (or any of its aliases) or via Z/XDC's Startup Panel in ISPF, it is easiest to allocate your concatenation to ddname **TASKLIB.**

Otherwise, allocate your concatenation to JOBLIB, STEPLIB or ISPLLIB (if running under ISPF).

- 2) **//xxxRENT8 DD DUMMY:** An alternative method can be used for debugging sessions that are started by the **XDCCALL** utility. (This includes sessions started via **options 1** or **2** of Z/XDC's Startup Panel in ISPF.)

  - In the batch, just add a **//xxxRENT8 DD DUMMY** card to your JCL.

  - In TSO, issue the command **TSO ALLOCATE DDNAME(xxxRENT8) DUMMY** prior to starting your debugging session. (But see (3) below.)

XDCCALL will then take steps to cause all of your programs to be loaded into fetch protected key 8 storage:
  - Regardless of reentrancy.
  - Regardless of whether or not the load library is APF authorized.
  - And regardless of your System's CSVRENTSP252 setting (from the DIAGxx member of PARMLIB).

(Note, XDCCALL's method for accomplishing this is to cause TCBTCP=1 to be set.)

Using //xxxRENT8 will not work for debugging sessions that are started without the help of XDCCALL.

- 3) **Z/XDC's Startup Panel in ISPF:** When using the Startup Panel, it has a setting that will cause it to allocate the //xxxRENT8 (and //xxxREFR8) ddnames for you. Look for the field captioned:
    Should RENT and REFR pgms be loaded into key-8 Storage?

- 4) A fourth alternative is to use Z/XDC's **SET AUTH** command (issued from an external, authorized debugging session) to make your debugging session authorized **without affecting** the authority level of the program you are debugging. See HELP COMMANDS SET AUTH for more information.

Debugging Reentrant Programs While Authorized

When Z/XDC is invoked authorized, it always sets itself to run with execution key 0. Accordingly, when authorized, it doesn't matter whether reentrant programs are loaded into fetch protected key 8 storage or fetchable key 0 storage. Either way, you will have no problems setting breakpoints or performing zaps.

[If your program is located in **store protected** storage, then you may have to issue

a **SET ZAP SPROT** command before being able to set breakpoints.]

In fact, for authorized programs, it is **not** a good idea to do anything that would artificially put reentrant load modules into key 8 storage. This is because then the modules would be in an environment different from that in which they were intended to run. Also, if, for example, your program issued SPKA instructions to change its execution key, and if the code were located in fetch protected storage, then it would probably suffer an 0C4 ABEND, instantly.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

STORAGEKEYS - An explanation of the "key controlled protection" mechanism used by z/OS to control accesses to storage.

Help DEbugging REEntrant Storagekeys

z/OS has many mechanisms by which it can prevent programs from reading from or storing into an area of storage; however, the mechanism that matters most regarding the setting of breakpoints is called "key controlled protection". Briefly, this kind of protection works as follows:

- Every page of storage has assigned to it a 4-bit numeric key whose value can range from 0 to 15. This is called the page's "storage key".

- The System's PSW, which controls the flow of execution through a program, has a 4-bit field called the "execution key".

- Generally, in order for a program to store into an area of storage, the program's execution key in the PSW must match the storage key of the page to be stored into; otherwise, a code-4 program check (0C4 ABEND) will occur. There are a couple of exceptions:

  - If a program is running with an execution key of 0, it can store into any page having any storage key. However, only an authorized program can run with its execution key set to 0.

  - If the storage being stored into has been assigned storage key 9, than ANY program, authorized or non-authorized, running with any execution key, can store into it. (This capability is used primarily for user data and programs running within CICS.)

  - If a program is running under the control of a key-mask permitting multiple keys, then that program can store into any storage whose storage key matches one of the keys specified in the key-mask. (For non-authorized programs, the key-mask usually specifies only keys 8 and 9.)

- Every page also has an additional bit called the "fetch protection bit". This bit controls whether or not a program can even reference (i.e. read) the page. This

bit works as follows:

- When the fetch protected bit is on, the page is said to be "fetch protected", and a program's execution key must match the page's storage key before a reference will be permitted. Exceptions:
  - A program running with an execution key of 0 can reference any page having any storage key.
  - Fetch protection does not apply to key 9 storage. Any program running under any key can both fetch from and store into key 9 storage.
- When the fetch protection bit is off, any program running with any execution key can fetch from the page.

Normally, when the System loads a program into "user key" storage, that storage will be key 8 and fetch protected. That means:

- Only programs running with execution keys 8 and 0 will be able to execute the code. They also will be able to change the code.
- If the program should suddenly change its execution key to a key other than 8 or 0 (as an authorized program might, via the SPKA instruction), it will immediately fail with an 0C4 ABEND (because the storage containing the code is fetch protected).

Normally, when the System loads a program into key 0 storage, that storage will be fetchable. That means:

- Any program running with **any** execution key will be able to execute the code without suffering an 0C4 ABEND.
- The program can be executed using any execution key. In particular, a problem state PSW, which usually runs with execution key 8, will be able to execute the key 0 code without failing.
- But said problem state PSW will not be able to execute any instruction that attempts to store into the code. The code is protected.

Help DEbugging REFreshable

If you're running Z/XDC on a z/OS operating system that's at release R1.8 and older, then you can skip this topic. It does not apply to you. However, if you're using a z/OS R1.9 or newer system, then read on.

Also, if you are running Z/XDC authorized, then you too can skip this topic. This topic only matters when Z/XDC is running non-authorized.

The Problem

1) When load modules and program objects reside in key 0 storage and Z/XDC is running **non-authorized,** debugging is highly crippled.

2) Starting with z/OS release 1.9, it began to be possible to cause the System to load **refreshable** modules into key 0 storage, thus severely limiting Z/XDC's usefulness when debugging non-authorized programs.

3) Fortunately, there is a System supported method for causing refreshable programs to be loaded into key 8 storage regardless of their refreshability attribute. Read on.

The Details

Starting with z/OS R1.9, System support became available for loading modules having the REFR attribute into fetchable **key 0** storage (instead of fetch protected user key storage). This feature is called **REFRPROT,** and it can be turned on or left off System-wide according to PARMLIB settings. For more information, see "z/OS MVS Initialization and Tuning Reference" (SA22-7592, -15 and newer).

In addition, the REFRPROT setting (when on) can be overridden at the individual task level. So when REFRPROT is active System-wide, it still can be deactivated for individual tasks such that it will have no affect upon program loading occurring under only those tasks.

Notes:

- z/OS will automatically propagate the REFRPROT override setting from parent tasks to daughter tasks at ATTACH time.

- z/OS does not support activating REFRPROT for a given task when REFRPROT is disabled System-wide.

When REFRPROT is active for a given task, then all refreshable (REFR) load modules and program objects loaded into storage (LINK, LOAD, ATTACH and XCTL) by code running under those tasks will be loaded into key 0 storage, **regardless** of whether or not it was loaded from an APF authorized, library, and **regardless** of whether or not the RENT attribute also is on.

On the other hand, when REFRPROT is disabled for a given task, then a load module's or program object's refreshable attribute (REFR) will have no affect whatsoever over the storage into which the module is loaded. (If a module **does** get loaded into key 0 storage, the in won't be REFR's fault. Instead, it will likely be due to the **RENT** attribute. See HELP DEBUGGING REENTRANT for more information.)

When Z/XDC is running authorized, then whether or not a module is loaded into key 0 storage doesn't matter. That's because Z/XDC will still be able to set its breakpoints as necessary, regardless.

But when Z/XDC is running, non-authorized, key 0 storage presents a real problem! Without taking particular steps, you will not be able to use Z/XDC to set breakpoints, to step through execution (tracing), or to zap any part of the load module.

When Z/XDC is being used to debug a non-authorized program, you need to find a way to cause that program to be loaded into unprotected storage (user key storage) instead of protected storage (key 0 storage). That's because Z/XDC runs under the same execution key as does your program, and that execution key makes it impossible

for Z/XDC to store into any storage other than storage whose storage key matches the execution key. In other words, for integrity and security reasons, Z/XDC is designed to inherit the same limitations that the system imposes upon your program.

Possible Solutions

Here are some ways to either resolve or get around this problem.

Remove the REFR Attribute: You can relinkedit your programs without the REFR attribute. That will, of course, eliminate the REFRPROT issue entirely.

But it may still be the case, under certain circumstances, for your module still to be brought into key 0 storage, but now that will have to do with the RENT attribute, not the REFR attribute. See HELP DEBUGGING REENTRANT for a discussion of that not so little wrinkle.

Nevertheless, if you (like me) have gotten into the habit of routinely specifying both the RENT **and REFR** attributes for no particular reason other than that your module was reentrant, you might consider changing that habit and dropping REFR altogether.

//xxxREFR8 DD DUMMY: If you are using the xxxCALL program to start your debugging session (see HELP XDCCALL), then you can add a **//xxxREFR8 DD DUMMY** card to your program's JCL. This will cause the xxxCALL utility to start off the debugging session with REFRPROT disabled (TCB_REFRPROT_OVERRIDE=1) for both your main task and all of its subtasks. (z/OS automatically propagates the REFRPROT override setting at subtask ATTACH time.)

!!!WARNING!!! It generally is a bad idea to use //xxxREFR8 when debugging authorized programs, so it is not recommended for use when using the xxxCALLA utility. For details, see HELP DDNAMES REFR8.

SET REFRPROF Command: For any debugging session, regardless of whether or not it was started under the xxxCALL utility, you can use Z/XDC's SET REFRPROT command to enable or disable REFRPROT within various scopes. See HELP COMMANDS SET REFRPROT for details.

Run Authorized: You can use the xxxCALLA utility to run your debugging session authorized. then the REFRPROT setting simply won't matter.

SETPROG NOREFRPROT System Command: If you are running on your own private sandbox system, you could use the SETPROG NOREFRPROT System Operator command to turn off REFRPROT System-wide. See IBM's "z/OS MVS System Commands" (SA22-7627, -16 and newer) for details.

Poisoning the Concatenation: Unfortunately (and unlike the issues raised by the RENT attribute), "poisoning the concatenation" is **not** going to work for defeating the effects of the REFRPROT facility. That's because the nature of the library from which the module is loaded is irrelevant. It can be APF authorized or not, the REFRPROT facility just doesn't care!

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

STORAGEKEYS - An explanation of the "key controlled protection" mechanism used by z/OS to control accesses to storage.

Help DEbugging REFreshable Storagekeys

z/OS has many mechanisms by which it can prevent programs from reading from or storing into an area of storage; however, the mechanism that matters most regarding the setting of breakpoints is called "key controlled protection". Briefly, this kind of protection works as follows:

- Every page of storage has assigned to it a 4-bit numeric key whose value can range from 0 to 15. This is called the page's "storage key".

- The System's PSW, which controls the flow of execution through a program, has a 4-bit field called the "execution key".

- Generally, in order for a program to store into an area of storage, the program's execution key in the PSW must match the storage key of the page to be stored into; otherwise, a code-4 program check (0C4 ABEND) will occur. There are a couple of exceptions:

  - If a program is running with an execution key of 0, it can store into any page having any storage key. However, only an authorized program can run with its execution key set to 0.

  - If the storage being stored into has been assigned storage key 9, than ANY program, authorized or non-authorized, running with any execution key, can store into it. (This capability is used primarily for user data and programs running within CICS.)

  - If a program is running under the control of a key-mask permitting multiple keys, then that program can store into any storage whose storage key matches one of the keys specified in the key-mask. (For non-authorized programs, the key-mask usually specifies only keys 8 and 9.)

- Every page also has an additional bit called the "fetch protection bit". This bit controls whether or not a program can even reference (i.e. read) the page. This bit works as follows:

    - When the fetch protected bit is on, the page is said to be "fetch protected", and a program's execution key must match the page's storage key before a reference will be permitted. Exceptions:

        - A program running with an execution key of 0 can reference any page having any storage key.

        - Fetch protection does not apply to key 9 storage. Any program running under any key can both fetch from and store into key 9 storage.

    - When the fetch protection bit is off, any program running with any execution key can fetch from the page.

Normally, when the System loads a program into "user key" storage, that storage will be key 8 and fetch protected. That means:

- Only programs running with execution keys 8 and 0 will be able to execute the code. They also will be able to change the code.

- If the program should suddenly change its execution key to a key other than 8 or 0 (as an authorized program might, via the SPKA instruction), it will immediately fail with an 0C4 ABEND (because the storage containing the code is fetch protected).

Normally, when the System loads a program into key 0 storage, that storage will be fetchable. That means:

- Any program running with **any** execution key will be able to execute the code without suffering an 0C4 ABEND.

- The program can be executed using any execution key. In particular, a problem state PSW, which usually runs with execution key 8, will be able to execute the key 0 code without failing.

- But said problem state PSW will not be able to execute any instruction that attempts to store into the code. The code is protected.

Help DEbugging RMode64

Starting with z/OS R2.3, the Operating System added support for placing load modules into 64-bit storage. Starting with Maintenance update Z22-1707B, Z/XDC z2.2 began supporting that change.

But Z/XDC's support goes a bit further than IBM's. Z/XDC's support for Privately Loaded modules was also updated to support above-the-Bar locations, and that support works just fine even on z/OS Systems older than R2.3.

Help DEbugging Srbmode

Z/XDC can be used to debug SRB mode routines. Just set up Z/XDC as the SRB's newest

FRR. Do this in the same way that you would for your own FRR, but designate Z/XDC's normal entry address (on the **IEAMSCHD** macro) in place of your own FRR routine. (If using the SCHEDULE macro, store Z/XDC's address into the SRBFRRA field prior to issuing the SCHEDULE macro.)

Z/XDC's normal entry point usually is named **XDC,** but when Z/XDC is installed under a clone name ("V31", for example), then its entry point name will be the clone name. (For information about renaming Z/XDC, see HELP XDCCLONES.)

In almost all cases when Z/XDC is intended to be used as an FRR, its **XDC** and **XDC31** load modules must be installed into common storage (the PLPA or the Dynamic Link Pack Queue).

When the **XDC** load module resides in common storage, you can use our **#XDCLOC8** macro to find its entry address. (This is called Z/XDC's **Quick Locate** support.) In constrained environments, where use of z/OS's LOAD macro is prohibited, the #XDCLOC8 macro can be used instead. For more information, see HELP MACROS #XDCLOC8.

When Z/XDC is used to debug SRB routines, Z/XDC's process splits into two phases: a Local Phase and a Remote Phase:

- The Local Phase runs as an FRR within the SRB mode environment. Its job is to intercept an ABEND or breakpoint, extract information about the ABEND, pass that information to the Remote Phase running elsewhere, wait for the Remote Phase to complete its processing, then resume or terminate the SRB routine.

- The Remote Phase runs as an IRB in a task mode environment under some task of Z/XDC's choosing. Its job is to send displays to the user and receive and execute commands from the user.

For a more detailed description of this two phase process, see HELP DEBUGGING FRR.

The Remote Phase runs under what Z/XDC refers to as a **proxy task.** Whenever a proxy task is in control:

- The phrase **SRB PROXY** is displayed in the upper left-hand corner of your terminal's display screen.
- The **LIST TASKS** command labels the proxy task with the phrase **SRB-FRR PROXY.**

For more detailed information, see HELP DEBUGGING FRR.

Inherent Dangers

In z/OS, SRB mode environments can be both highly constrained and highly sensitive. There will be circumstances where interactive debugging can be conducted safely, and you will find Z/XDC to be an extremely valuable tool there. But there also will be environments where use of Z/XDC will simply be **too dangerous** or even not possible! So you will have to learn from experience when Z/XDC can be used and when it can't.

The causes giving rise to these dangers, and the considerations you have to think about and the security protections that are available are discussed at length in **HELP DEBUGGING LOSTLOCKS.** Read that topic for more information.

Related Commands

When debugging SRB routines, Z/XDC's full range of commands can be used to assist in the process. However, the following commands offer explicit SRB related features:

- **LIST SSRBS**
- **LIST SRBS**
  This command generates a display describing all suspended SRBs located an any specified address space.

  It also regenerates the following automatic equates:

  - **SSRB#nnn**
    These equates label the SSRBs from which the **LIST SSRBS** command's display is generated.

  - **SSRX#nnn**
    If you are running on a z/OS R1.13 or newer system, then these equates label the SSRXs from which the **LIST SSRBS** command's display is generated.

  - **SSRBEP#nnn**
    These equates label the entry points of the SRB routines described by the **LIST SSRBS** display.

- **LIST registersetname ssrbaddress**
  These commands display the various register sets that are associated with any desired suspended SRB in the system. The register sets that can be displayed are:
  - General registers
  - Access registers
  - Floating point registers
  - Control registers

- **LIST registername ssrbaddress**
  These commands display various individual registers that are associated with any desired suspended SRB in the system. The types of registers that can be displayed are general, access, floating point and control.

- **LIST PSW ssrbaddress**
  This command displays the resume PSW for any desired suspended SRB in the system.

- **LIST LSTACK ssrbaddress**
  This command displays either the linkage stack or linkage stack entries for any desired suspended SRB in the system.

For all of the above commands. the **ssrbaddress** operand can be the address either of an SSRB or of an **SSRX.** Note that any of the SSRB#n and SSRX#n equates generated by the **LIST SSRBS** command can be used for the **ssrbaddress** operand.

Also, all of these commands (like all other Z/XDC commands) can be used both during SRB mode debugging and during normal task mode debugging.

dump/XDC Support for SRB Mode

Of course, when dump/XDC is used to debug a dump that was captured during SRB mode execution, **none** of the "Inherent Dangers" discussed above apply. The dump, after all, is dead (not live, running code).

For more information about dump/XDC's support of SRB mode dumps, see **HELP DUMPS EXECUTIONTHREAD SRBMODE.**

Help DEbugging TSo

Z/XDC can be used to debug normal programs and TSO commands running both within and outside of ISPF. Either way, Z/XDC provides a powerful fullscreen display interface, session profiling support, and session logging and scrolling support.

This topic discusses starting Z/XDC from TSO's **READY** prompt. For starting Z/XDC from the **Z/XDC Startup Panel** in ISPF, see HELP DEBUGGING ISPF.

The easiest way to start a debugging session in nearly any environment, including in TSO, is to use the **XDCCALL** command or any of its aliases. Briefly, XDCCALL can be used as described below. For complete information, see HELP XDCCALL.

Under certain conditions, you may have to modify your program in order to use Z/XDC to its fullest capabilities. See **HELP DEBUGGING GETTINGSTARTED** for more information.

XDCCALL and its aliases (XDCCALLA, XDCCMD, and XDCCMDA) provide a high level interface to Z/XDC. They create a debugging environment within which a normal program or TSO command can be executed. XDCCALL (and its aliases) accepts information about a program's name, its parameters if any, its load library(s), and its authorization level, and then it attaches that program as a subtask and establishes Z/XDC as an ESTAI type recovery routine for the subtask. (ESTAI routines are very similar to ESTAEs. They differ only in issues of ownership and in program retry rules. See **HELP EXECUTIONLEVELS** for more information.)

Invoking XDCCALL[A] or XDCCMD[A]

To use XDCCALL (and friends), proceed as follows:

- If the program that you want to debug resides in a library that is not within your TSO session's "load module search order" (e.g. is not in a //STEPLIB or linklist library), then before issuing the **XDCCALL** command you must first allocate that library to ddname **//TASKLIB.** Example:

  ```
  ALLOCATE FILE(TASKLIB) REUSE SHR DSN(LOADLIB)
  ```

- **XDCCALL** has three alias names: **XDCCALLA, XDCCMD** and **XDCCMDA.** You must start your debugging session using one of these four names depending upon the kind of program that you are debugging:

  - If your program needs to run authorized, then use either **XDCCALLA** or **XDCCMDA.**

    - If your program is a TSO command that expects R1 to point to a CPPL (Command Processor Parameter List: a TSO control block), then use either **XDCCMD** or **XDCCMDA.**

    - If your program expects R1 to point to a standard PARM field, then use **XDCCALL** or **XDCCALLA.**

    - If your program does not care what R1 points to, then it is irrelevant whether you use:
        - **XDCCALL** vs. **XDCCMD** [for problem state]
        - **XDCCALLA** vs. **XDCCMDA** [for authorized]

- When you issue whichever command you select, the first operand must be the name of the program or TSO command that you want to debug.

- If your normal program needs PARM field data, or if your TSO command needs operands, then type that data or those operands following the program/command name. By the time your program/command starts execution, XDCCALL/XDCCMD will have reformatted the PARM field data or command operands appropriately.

Examples

The following examples show how to use XDCCALL and its aliases to start debugging sessions against various familiar programs. The program names were chosen, not because you would actually want to debug them, but to illustrate the ∗kinds∗ of programs that are appropriate in the context of the examples.

XDCCALL IEBUPDTE NEW
This starts a debugging session against the IEBUPDTE program with "NEW" being passed to the program as PARM field data. This causes IEBUPDTE to execute in an environment that is similar to that which would be created by the following JCL statement: // EXEC PGM=IEBUPDTE,PARM=NEW

XDCCALLA IEBCOPY SIZE=1024K
This starts an authorized debugging session against the IEBCOPY program with "SIZE=1024K" being passed to the program as PARM field data. (Note that IEBCOPY needs to execute authorized, Why? Well, that’s a long and sad story.) This causes IEBCOPY to execute in an environment that is similar to that which would be created by the following JCL statement: // EXEC PGM=IEBCOPY,PARM=’SIZE=1024K’

XDCCMD LISTD ’SYS1.PROCLIB’ MEMBERS
This starts a debugging session against TSO’s LISTD command with the string **LISTD ’SYS1.PROCLIB’ MEMBERS** passed to the command via a CPPL.

XDCCMDA LISTBC
This starts an **authorized** debugging session against TSO’s **LISTBC** command with the string **LISTBC** passed to the command via a CPPL.

The Program Environment created by XDCCALL (and friends)

Once you have issued the XDCCALL command, a debugging session will start, and you will be presented with a fullscreen display with execution interrupted at a branch instruction that is just about to pass control to your program’s entry point. The

registers will be set to the following standard values:

R15 - If your program starts up in either 24-bit or 31-bit addressing mode, then R15 points to your program's entry address. If your program starts up in 64-bit addressing mode, then R15 contains the value **X'FFFFF00x'** (where **x** can be 0, 2 or 4. For more information, refer to IBM's **z/OS MVS Programming: Assembler Services Guide** [SA22-7605], and then search for the string **FFFFF002).**

R14 - Points to a return address that your program can use to terminate.

R13 - Points to the system provided save area buffer pointed to by the current TCB's **TCBFSA** field. This buffer has the following characteristics:
- It is 144 bytes long.
- So it is long enough to contain either a standard, 72-byte register save area, or a 144-byte, F4SA (format-4) register save area.
- Initially, the 144-byte save area buffer has been entirely zeroed by the system.

R1 - If XDCCALL or XDCCALLA was used, then R1 points to a PARM field. If XDCCMD or XDCCMDA was used, then R1 points to a CPPL.
- All other registers contain random values. They are **not** pre-zeroed, so your program must not depend on them being zeroed.

PSW - Points to your program's entry point.

Ok Cool! Your Session has Started... NOW WHAT!???

At this point, you may want to issue the following typical commands:

- **MAP R15?**
  This causes Z/XDC to read at least a module map for your program. This informs Z/XDC where, within your load module, your csects are located. Also, if you have created a csect map for your program, then that will be read as well. See **HELP MAPS** for more information.

- **S Q R15?**
  This defines your program's entry csect as a "default" csect. This gives meaning to address expressions that start with a dot. Examples: ".+2C", ".TOPLOOP-3B", and just ".". (See **HELP ADDRESSING PARSERS ASM RESOLUTION** for more information.) Note that the **SET QUALIFIER** command cannot be issued unless a **MAP** command is issued first.

- **FORMAT R15?**
  This displays the code and data located at your program's entry point. If Z/XDC formats the display poorly, then you need to create a csect map to give Z/XDC's formatter better guidance regarding what is data and what is instructions. Alternatively, you can use Z/XDC's **EQUATE** command to manually define labels with attributes that help to guide the formatter properly. See **HELP COMMANDS EQUATE** for more information.

- You may want to use the **DMAP** and **USING** commands to read and assign to appropriate storage locations dsect maps of IBM control blocks and of your own private control blocks. See **HELP COMMANDS DMAP** for more information.

- At this point you can use Z/XDC's various breakpointing commands to set traps both in the current load module and even in load modules that have not yet been brought into storage (using deferred breakpoints). See **HELP BREAKPOINTS** for more information.

- Finally, use the **GO** command to let your program start executing. When it reaches a breakpoint or an abend, Z/XDC will receive control again and present its displays.

Help DEbugging Uss

This topic discusses using Z/XDC is a USS environment. (including a non-USS environment that has been 'Dubbed').

What is a USS environment?

A USS environment exists when a new task (and in most cases, a new aspace) is created using one of the following USS requests (as documented in the USS Assembler calls manual or its C/C++ equivalent):

- attach_exec (BPX1ATX, BPX4ATX)
- attach_exec_mvs (BPX1ATM, BPX4ATM)
- fork (BPX1FRK, BPX4FRK) or (C/C++) fork()
- pthread_create (BPX1PTC, BPX4PTC) or (C/C++) pthread_create()
- spawn (BPX1SPN, BPX4SPN) or (C/C++) spawn(), spawnp(), spawn2(), spawnp2()

What is Dubbing?

Making a USS request as documented in the USS Assembler calls manual or its C/C++ equivalent in itself may cause a task to be **dubbed** (a z/OS term that means that the task has been made a USS process and assigned a process id) as a process, but does not cause that task to actually be executing as a USS process. Only the USS requests listed above do that.

Z/XDC can already be used to debug tasks that have been **dubbed** as USS processes. This topic is mainly concerned with debugging actual USS processes, created using one of the USS requests listed above. Tasks that have been dubbed are also mentioned where relevant.

The USS= **operand**

This topic is covered fully in **HELP DEBUGGING USS OPERANDS USS=**

The DUB= **operand**

This topic is covered fully in **HELP DEBUGGING USS OPERANDS DUB=**

USS names

This topic is covered fully in **HELP DEBUGGING USS NAMES**

Using Z/XDC with USS

This topic is covered fully in **HELP DEBUGGING USS XDC**

Phased Implementation

This topic is covered fully in **HELP DEBUGGING USS IMPLEMENTATION**

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

Command Operands	- Some Z/XDC commands have USS-related operands. These operands are described in **HELP DEBUGGING USS OPERANDS**
Implemenation	- The phased implementation.
Names	- USS-related names.
Z/XDC	- Using Z/XDC.

Help DEbugging Uss Operands

Several Z/XDC commands have had new USS-related operands added. They include the **USS=** and the **DUB=** operands.

The USS= operand is explained fully in **HELP DEBUGGING USS OPERANDS USS=**

The DUB= operand is explained fully in **HELP DEBUGGING USS OPERANDS DUB=**

Help DEbugging Uss Operands Uss=

The USS= **operand**

Several Z/XDC commands have the **USS=** operand on them.

If this operand is supported by a command, it can be specified **anywhere** on that command, and is **always** parsed and processed before other operands.

This operand allows you to specify how Z/XDC is to process the following:
- Sometimes Z/XDC needs to parse some operands on a command differently when **USS=YES** is in effect.
- Z/XDC needs to know how to process operands on a command that are ambiguous in a USS environment, for example a dataset or file name.

The **USS=** operand has the following values:
- PROFILE - Use the profile USS setting. (this is the default if the USS= operand is omitted).
- NO - Operands are processed 'traditionally'.
- YES - Operands are processed using rules that apply in a USS environment.
- IFUSS - See below.

The profile USS field can be set to the following values:
- NO - operands are processed 'traditionally'.
- YES - operands are processed using rules that apply in a USS environment.
- IFUSS - See below. (This is the default value in the profile).

If the operand value is **IFUSS** or the operand value is **PROFILE** and the profile value is **IFUSS** the following rules apply:
- the in-effect value depends on whether or not FASM (Foreign Address Space Mode) is in effect:
  - If not in FASM and the current task is not a formal USS process then we act as if **USS=NO** is in effect otherwise we act as if **USS=YES** is in effect.
  - If in FASM and the FASM aspace is not a USS aspace then we act as if **USS=NO** is in effect otherwise we act as if **USS=YES** is in effect.

Help DEbugging Uss Operands Dub=

The DUB= **operand**

Several Z/XDC commands have the **DUB=** operand on them.

If this operand is supported by a command, it can be specified **anywhere** on that command, and is **always** parsed and processed before other operands.

This operand allows you to specify how Z/XDC is to process in the following case:
- When Z/XDC needs to perform an operation that could result in the current task being 'dubbed' (a z/OS term) and the task is not already 'dubbed' then the value of this operand is relevant.

The **DUB=** operand has the following values:
- PROFILE - use the profile DUB setting (this is the default if the DUB= operand is omitted)
- NO - Dubbing is not allowed. If dubbing is needed, the command will be terminated with a message.
- YES - Dubbing is allowed. If dubbing is needed, dubbing will be done.
- ASK - If dubbing is needed, a question will be asked.

The profile DUB field can be set to the following values:
- NO - dubbing is not allowed. If dubbing is needed, the command will be terminated with a message.
- YES - Dubbing is allowed. If dubbing is needed, dubbing will be done.
- ASK - If dubbing is needed, a question will be asked. (This is the default value in the profile)

Note that Z/XDC **always** acts as if **DUB=YES** is in effect in the following cases:
- A READ file is active (and not paused).
- The command has been issued by a REXX procedure.
- The command is in an initial command list.

Help DEbugging Uss Implementation

Phased implementation

Full USS support is being introduced to Z/XDC in phases.
Currently, this support is in phase 2. (lower-numbered phases have also been done)

Phase 2 support in Z/XDC for USS has the following changes:
- The messages issued at the start of a debugging session or when Z/XDC is (re)entered due to an ABEND, or breakpoint will indicate the the current task is a USS process or that the current home address space has been built to execute USS processes. Note that these messages are also issued by the **LIST MSGS** command.
- The **LIST TASKS** command will indicate that a task is a USS process or thread, or has been dubbed.
- The new **LIST CWD** command displays the USS Current Working Directory (CWD).
- The new **LIST PID** command displays the USS Process ID.
- The **LOAD** command supports the loading of program objects from a USS file.
- The **DELETE MODULES** command supports the deletion of program objects that were loaded from a USS file.
- the **MAP** command supports the mapping of program objects in a USS file.
- the **DMAP** command supports the loading of DSECT maps from a USS file.
- The **LIST ASID** command will indicate that the address space is or is not a USS address space.
- The **LIST USS** command can be used to display USS information about the current task.
- The **SET ASID** command will indicate that the set-to address space is or is not a USS address space.
- if a program is executed using the **XDCCALL[A]** program, If dubbing has occurred, the environment is undubbed before exiting, to avoid the **BPX018I** message being produced by the system.
- the user's profile has the following USS-support-related fields stored on it:
  - the USS setting
  - the DUB setting

  The settings can be displayed and changed using the following:
  - the **LIST USS** command.
  - the **SET USS** command.
  - the **LIST DUB** command.

- the **SET DUB** command.
- The profile menuing system.

Note that while the various breakpointing and tracing commands (including hooks) can be used to set breakpoints (etc.) in path name programs, deferred breakpoints and hooks **cannot** currently be set in path name programs.

Phase 1 support in Z/XDC for USS has the following changes:

- The **LIST TASKS** command will show the file name of a program loaded by USS instead of ***PATHNAM**
- The **LIST RBS** command will show the file name of a program loaded by USS instead of ***PATHNAM**
- The **LIST PGMS** command will show the path name of a program loaded by USS. (the short name will be displayed as '-'). This command has also been changed to allow several operands to be USS file names.
- A storage location that is identified as being within a program loaded by USS has the name of the USS file that the program was loaded from instead of ***PATHNAM**
- The syntax of address expressions has been changed to accept a USS-loaded program's file name provided that it has no embedded blanks or special characters.
- An address expression may start with a quoted USS file or path name to allow that USS file or path name to contain characters that are not permitted or have a special meaning in an address expression (such as a period, dash, or plus sign).
- in Z/XDC, a quoted value may be enclosed in apostrophes (or tic marks) (') (as before) or literary quotes ("). The provision of an extra quote character gives increased flexibility when specifying command strings on (for example) the **TRACE** command.
- (Note that the rules for the the **LIBRARYLISTS** command **have not** changed.)
- in Z/XDC, an **escape character** can be used in several contexts to suppress any special meaning assigned to the next character. The **Escape character** is the backslash (\).
- A new builtin function ('LMOD(...)') can be used in address expressions to allow the specification of an arbitrary path or file name containing blanks and/or special characters.

Help DEbugging Uss Names

USS Names (and ambiguous names)

In USS, names are different to names used elsewhere in z/OS:

- A USS-related name can contain mixed-case characters, and the case of those characters are significant (i.e., the name is not folded to a single case).
- A USS-related name may be much longer than a **traditional** name.
- A USS-related name is variable length.
- A USS-related name may contain characters that are not allowed in a **traditional** name.

Z/XDC handles USS-related names correctly. In some cases, Z/XDC truncates a name to a reasonable length.

in many cases, a name may be ambiguous. This is because an unquoted name (maybe including a member name) looks like both a dataset name or a USS path name. (this is

because the **.()** characters are all valid USS path name characters)
(Note that in some cases, a single-level name (i.e., no fullstops) was traditionally treated by Z/XDC as a ddname. However, under USS, a single-level name is quite common, and often indicates a file in the Current Working Directory (CWD) - ambiguity still applies)
Ambiguity may be resolved in several ways:

- A command operand may only accept one or the other type. Note that this rule applies in the case where the operand is positional or indicated by a keyword
- A quoted value is always treated as a USS path or file name.
- If the **USS=NO** operand is specified on the command, (when allowed) or **USS=NO** is in effect an unquoted value is always treated as a dataset name.
- If the **USS=YES** operand is specified on the command, (when allowed) or **USS=YES** is in effect an unquoted value is always treated as a USS path name.
- In some cases, for compatibility with previous releases of Z/XDC, an unquoted value is treated as a dataset name.

USS Path and file names

When Z/XDC needs to save a path name in a control block (for example, the path name area pointed to by the CDX, in turn pointed to by the CDE or LPDE) the path name is copied. Z/XDC only keeps the 255 **rightmost** characters of the path name, and only keeps the 255 **leftmost** characters of the file name. If truncation occurs, an error is indicated.

When a path or file name must be supplied to Z/XDC, the value is limited to **255** characters. In addition, the overall length of an Z/XDC command is limited to **255** characters, which may limit the length of a USS path or file name operand.

Once Z/XDC has validated and accepted the USS path or file name, any internal expansion is only limited by the USS limit itself, which is currently **1023** characters for a path, and **255** characters for a file name or a segment of a path (between '/' characters).

To supply a path name which is longer than Z/XDC will accept, you can make use of **Symbolic Links** in a path name.
For example:

- Assume you have defined **MLN** as a symbolic link for **a/very/long/pathname** This can be done using the following USS command:
  ln a/very/long/pathname MLN
- You want to specify a path name of **a/very/long/pathname** on a command operand (in this example the operand name is X=)
- You can specify **X=MLN** for the command operand.

If a USS path or file name contains any of the following characters in it, it **must** be quoted to ensure that Z/XDC treats those characters as part of the USS path or file name:

- + (plus sign)
- - (minus sign or dash)
- * (asterisk)
- ? (question mark)
- ((left parenthesis)
-) (right parenthesis)
- . (full stop or period)
- (blank)
- ¢ (cent sign)

- ` (back accent)
- ~ (tilde)
- ’ (apostrophe or tic) (Note that if the quoted string is surrounded by this character it must be doubled-up in the quoted string)
- " (literary quote) (Note that if the quoted string is surrounded by this character it must be doubled-up in the quoted string)
- \ (escape character) (see below)

If the first 2 characters of a path name are ~/ (tilde-slash) then, as explained below, they may get replaced.

USS path names and the TILDE character (~)

If a path name commences with the characters **~/** the **~** is (maybe) replaced with the user’s home directory.

Note that the tilde (~) is one of the characters that MUST **be quoted in a USS path name. If the above sequence is used the path name** must **be quoted.**

The following things must happen:
1: Dubbing is required.
2: The user’s home directory name is retrieved. (If this fails, Possibly the user is not allowed to use USS)
3: The user must have a home directory.
4: The home directory must be absolute (i.e., start with a ’/’).
5: The new path name must not be too long.

If any of the above things fail, the replacement is not done and the tilde (~) is left in place.

Note that the general UNIX shell processing where a sequence of **~userid/** causes replacement of **~userid** with the named user’s home directory is not done - the sequence is left in place.

USS Path name variables

In most cases, a USS path name may contain variables. Command operands that do not allow this will be marked. A variable that is specified in a USS path name will be replaced by that variable’s value.

A variable is indicated by the sequence ∗v (asterisk variable) where the variable ’name’ is a single character as follows:

∗	an asterisk(∗)
d	The current date in the format dyymmdd where the first ’d’ is a lowercase ’d’ and the remaining characters are decimal digits.
D	The current date in the format Dyymmdd where the first ’d’ is an uppercase ’D’ and the remaining characters are decimal digits.
i	The process Id in the format pppppppp where the ’p’s are hex characters 0-9 and a-f. The process id is represented as a hexadecimal number in lower case.
I	The process Id in the format pppppppp where the ’p’s are hex characters 0-9 and A-F. The process id is represented as a hexadecimal number in upper case.
j	The jobname in lower case (trailing blanks are deleted, and the resulting value will be 1 to 8 characters lomg). If the jobname is not available,

'notfound' will be used.

J The jobname in upper case (trailing blanks are deleted, and the resulting value will be 1 to 8 characters lomg). If the jobname is not available, 'NOTFOUND' will be used.

o The region's owner id in lower case. (trailing blanks are deleted, leading and trailing '+' characters are removed (so '+master+' will have the value 'master'), and the resulting value will be 1 to 8 characters long). If the owner id is not available, the region's jobname (in lower case) will be used (as if 'j' was specified instead of 'o').

O The region's owner id in upper case. (trailing blanks are deleted, leading and trailing '+' characters are removed (so '+MASTER+' will have the value 'MASTER'), and the resulting value will be 1 to 8 characters long). If the owner id is not available, the region's jobname (in upper case) will be used (as if 'J' was specified instead of 'O').

p The region's TSO prefix in lower case. (trailing blanks are deleted, and the resulting value will be 1 to 8 characters long). If the TSO prefix is not available (including the case where this region is not a TSO region) or the TSO prefix is invalid the region's jobname (in lower case) will be used (as if 'j' was specified instead of 'p').

P The region's TSO prefix in upper case. (trailing blanks are deleted, and the resulting value will be 1 to 8 characters long). If the TSO prefix is not available (including the case where this region is not a TSO region) or the TSO prefix is invalid the region's jobname (in upper case) will be used (as if 'J' was specified instead of 'P').

t The current time in the format thhmmss where the first 't' is a lowercase 't' and the remaining characters are decimal digits.

T The current time in the format Thhmmss where the first 'T' is an uppercase 'T' and the remaining characters are decimal digits.

u The region's security system user id in lower case. (trailing blanks are deleted, and the resulting value will be 1 to 8 characters long). If the user id is not available the region's owner id (in lower case) will be used (as if 'o' was specified instead of 'u').

U The region's security system user id in upper case. (trailing blanks are deleted, and the resulting value will be 1 to 8 characters long). If the user id is not available the region's owner id (in upper case) will be used (as if 'O' was specified instead of 'U').

w The Current Working Directory (CWD)

x The current Z/XDC clone name (always 3 upper case characters)

X The current Z/XDC clone name (always 3 upper case characters)

y The current year in the format yyyyy where the first 'y' is a lowercase 'y' and the remaining characters are decimal digits.

Y The current year in the format Yyyyy where the first 'Y' is an uppercase 'Y' and the remaining characters are decimal digits.

USS and the Escape character

Within Z/XDC, the \ (escape character) can be used to **escape** any special meaning of the following character.

For example, in a wildcard string, a * or ? character normally means a wildcard. However using * or \? causes those characters to be treated literally, i.e., as themselves.

The preceding example is particularly relevant when a mask string could be a USS path or file name and the path or file name needs to contain * or ? (or \) characters. Escaping the instances of the actual characters would prevent them from

being treated as wildcard characters.

In a quoted string, in this example enclosed in ’ (apostrophes or tics) you can use the escape sequence \’ as an alternative to doubling up the apostrophes or tics if you want one in the quoted string.

In a USS path name, the variable designator (* - asterisk) may be escaped. In this case a literal asterisk is inserted into the generated path name and the following character (the variable name) is treated literally.

Help DEbugging Uss Xdc

Using Z/XDC in a USS environment

In a USS environment, Z/XDC can be used in the following ways:
- Using the **XDCUSS** program.
- Using Z/XDC’s **HOOK** command.
- Modifying an existing program or writing a new program.

Using the XDCUSS program

This section will be provided later.

Using the Z/XDC HOOK **command**

This section will be provided later.

Modifying an existing program or writing a new program

This section will be provided later.

Help DEbugging TIps

The following topics describe several tips and tricks that I or my customers have discovered over many years of using Z/XDC. If you have a tip of your own that you would like to see added here, then call me and tell me about it. If I do add it, then I will, of course, give due credit to you for giving me the idea.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

ABENDREASCD - Finding ABEND return codes and reason codes.
DUPLICATES - Mapping individual instances of duplicate load modules.
0C4 - Why inconsistent 0C4s may occur.

Help DEbugging TIps Abendreascd

When your program ABENDs and you go to look up the ABEND code in IBM's "System Codes" manuals (z/OS: SA22-7626, OS/390: GC28-1780, MVS/ESA 4.0 thru 4.3: GC28-1664, MVS/XA 2.2: GC28-1157, MVS/SP 1.3: GC38-1008), you will often find that the ABEND has several different possible "reason codes" associated with it, and you will have to discover what those codes are. Sometimes, those codes are conveniently located in R15 or (possibly) R0, and so all you need to do is check those registers.

Unfortunately, for some ABENDs the reason code is somewhat harder to find. This is because it will be located in a field of the SDWA instead of in a register, and generally the ABEND description in the IBM manual will not clearly state this. It may just say something like, "Check the reason code", but it won't say where to find that reason code.

When a reason code is contained in the SDWA, it will be found in a field named SDWACRC. But this field is not located precisely in the SDWA itself. Instead, it is located in one of the several sub-blocks that are chained from the SDWA, so unless you've memorized the specific chain of pointers that leads to the SDWACRC field ("@SDWA+170?+4?+2C"), finding that field can be rather tedious.

An easy way to solve this problem is to use the SDWAMAPS member of the XDCCMDS library. (The library's factory default name is hlq.XDCV31.XDCCMDS. Ask your Systems Programmer for its actual name at your Data Center.) This is a file of Z/XDC commands that can be used to read the dsect maps for the SDWA and all of its sub-blocks, and assign them to map the SDWA and the sub-blocks that describe the user program ABEND that has caused Z/XDC to receive control. Then all you have to do is use the FORMAT command to display the SDWACRC field. For more information, see HELP SCRIPTS OURSCRIPTS SDWAMAPS.

SPECIFIC INSTRUCTIONS:

READ dsname(SDWAMAPS)
This command reads and executes the SDWAMAPS member of the XDCCMDS library. Unfortunately, I cannot tell you here what the exact dsname of that library is, because that depends upon decisions made by the Systems Programmer who installed Z/XDC at your Data Center. So you will have to obtain that information from him. All I can tell you is that the library's dsname probably ends with ".XDCCMDS".

FORMAT .SDWACRC
This command shows the contents of the SDWACRC field. For some ABENDs, this will be the reason code associated with that ABEND.

Help DEbugging TIps Duplicates

There are many circumstances whereby z/OS might bring multiple copies of a load module into storage. The most common is issuing multiple LOAD macros against a load module that has not been marked (by the Binder) as being either reentrant (RENT) or serial reusable (REUS).

Unfortunately, Z/XDC does not have direct support for referring to multiple duplicate instances of a load module. (Appending a subscript, for example, is not yet supported for load modules.) Generally, references by name to a load module will be resolved to the first instance encountered in z/OS's standard search order. (See **HELP ADDRESSING PARSERS ASM MODULES.)** However, if a particular instance of the module has been mapped, then subsequent by-name references to that module will be resolved to the mapped instance (not necessarily the first instance).

Mapping a Particular Instance of a Load Module or Program Object

A load module's first instance can be mapped and referenced in the usual way:

```
MAP name.csect
SET QUALIFIER name.csect
FORMAT .label
etc.
```

A slightly different procedure must be used if you wish to map an instance of a module that is not the first instance:

1st: If a different instance of the module has already been mapped, then delete that map. This will avoid possible confusions later. Example: **DELETE MAPS name**

2nd: Display any location within the desired instance of the module. Here are some examples for doing that.

- If you know the virtual address of the desired copy, you can use either the **FORMAT** or **MAP** command to display or map that instance. Example: **FORMAT 265D4AC8**

- If you issue the **LIST PGMS name** command, a display will be produced showing the locations of all instances of the named module. You can then use either the **F** or **M** shortcut (typed into the Shortcut Entry Field at the left) to display or map the desired instance.

- If the PSW or a register points into the desired instance, then you can use it to either display or map that instance. Example: **FORMAT R12?**

- If you've mapped an instance of the desired module, then references via that map will resolve to that instance. Example:

```
MAP PSW?
SET QUALIFIER PSW?
FORMAT .3C8
FORMAT .TOPLOOP
```

3rd: If you haven't done so already, then map the displayed instance via a **MAP +0** command.

Warning: Z/XDC supports mapping multiple instances of a load module, but when multiple instances of the map are present, only the most recently loaded of those maps can be used. This is because there is no way to give unique names (via the MAP command) to multiple instances of a load module's map. So all older instances become unavailable for reference. Generally though, it is a good idea to keep only one instance of a map loaded at a time. Doing otherwise, can too easily lead to unnecessary confusion.

Using the DMAP Command to Map *ALL* Instances of a Load Module or Program Object

Now, while there is no way to use the MAP command to create uniquely named multiple instances of a load module map, there is a way to do this using the **DMAP** command. In fact, this is the kind of thing that is done all the time for dsect maps. The trick here is to realize that while load module maps are normally loaded by the MAP command, they can also just as easily be loaded by the DMAP command. The difference is, when the DMAP command is used, the load module map takes on all of the properties of dsect maps. In particular, multiple uniquely named copies can be created, and each copy can be independently assigned to storage via the **USING** command.

Mapping Multiples Example

The following is a sample procedure for using the DMAP command to map multiple instances of a load module.

LIST PGMS name
First, use the LIST PGMS command to produce a display of the load module and all of its instances. Write down the addresses of each instance that you care about. (You may have to take into consideration that the displayed addresses are the entry point addresses, not the load point addresses. When the two addresses do not coincide, you will have to adjust the following suggestions appropriately.)

LIST LKEDMAP name
Then, use the LIST LKEDMAP command to determine the length of the load module. Write that down too. (This will also tell you if the entry address and the physical starting address do not coincide.)

EQUATE name1st xxxxxxxx llllll
EQUATE name2nd yyyyyyyy llllll
EQUATE name3rd zzzzzzzz llllll
etc.
Then, use the EQUATE command to assign uniquely named equate labels to each instance that you care about. For the equate's address, use the actual hex address of the **PHYSICAL START** of each instance. As a nicety, also assign the load module's length to be the coverage length of the equate.

DMAP name. <-- The trailing dot is important here
Then, use Z/XDC's DMAP command to read the load module's module map as if it were a dsect.

DMAP name#1 name
DMAP name#2 name
DMAP name#3 name
etc.
Then use the DMAP command again to make as many clones of the module map as you need, with each clone having a different name. Among other things, doing this also avoids ambiguities and human confusion both with the original module map's name (if also loaded) and with the load module itself.

USING name#1 name1st FLOAT
USING name#2 name2nd FLOAT
USING name#3 name3rd FLOAT
etc.
Then, use the **USING** command to place each instance of the cloned module dsect map onto the corresponding instances of the load module itself. (The "FLOAT" operand is not necessary, but it does make it easy to automatically move the module map and dependent csect maps [see below] simply by redefining the underlying equate to a new location.)

DMAP name.csect
DMAP csect#1 csect
DMAP csect#2 csect
DMAP csect#3 csect
USING csect#1 name#1.csect FLOAT
USING csect#2 name#2.csect FLOAT
USING csect#3 name#3.csect FLOAT
Now do the same for csect maps. Use the DMAP command to read a csect map as if it were a dsect. Then use the DMAP command again to make clones of it. Then use the **USING** command to assign those clones to the appropriate locations within the appropriate instances of the module maps.

Labels within csects within specific instances of the load module can now be referred to as "csect#n.label". It is not necessary (and in fact not correct) to include the name of the module map into the reference.

Referring to just **.label** will refer into the instance of the csect map that was assigned to storage via the most recently issued **USING** command. If you wish it to refer to a different instance, then simply reissue the **USING** command corresponding to the desired instance, and then make the reference again. See **HELP COMMANDS USING** for more information about how the **USING** command controls the dsect search order.

Subtopics Index

For a comprehensive example, select the following topic. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

EXAMPLE - An annotated session log showing how to set up a series of maps for referencing duplicate load modules.

Help DEbugging TIps Duplicates Example

Below is a Z/XDC session log illustrating the process of setting up a series of maps for referencing duplicate load modules. **Study it carefully!** It contains commentary illustrating and explaining **additional** considerations.

load pds84

```
ENTRY    PRIMARY  REFRNCE ENTRY     USE   SUB    ATTRIBUTES
NAME     NAME     TCB/RB# ADDRESS  COUNT POOL KY (ATTR ATTR2 ATTR3-ATTRB)
PDS84             10  LLE 000CE000  1-0   251 8p 032240-12 JPA SYSLIB PML CDEX

ENTRY    PRIMARY  DFINING ENTRY     USE   SUB    ATTRIBUTES
NAME     NAME     QUEUE   ADDRESS  COUNT POOL KY (ATTR ATTR2 ATTR3-ATTRB)
PDS84             JPQ-3   000CE000    1   251 8p 032240-12 JPA SYSLIB PML CDEX
```

I've loading into storage the first instance of a nonreentrant nonreusable load module.

load pds84

```
ENTRY    PRIMARY  REFRNCE ENTRY     USE   SUB    ATTRIBUTES
NAME     NAME     TCB/RB# ADDRESS  COUNT POOL KY (ATTR ATTR2 ATTR3-ATTRB)
PDS84             10  LLE 00105000  1-0       8p 032240-12 JPA SYSLIB PML CDEX
PDS84             10  LLE 000CE000  1-0   251 8p 032240-12 JPA SYSLIB PML CDEX

ENTRY    PRIMARY  DFINING ENTRY     USE   SUB    ATTRIBUTES
NAME     NAME     QUEUE   ADDRESS  COUNT POOL KY (ATTR ATTR2 ATTR3-ATTRB)
PDS84             JPQ-3   00105000    1       8p 032240-12 JPA SYSLIB PML CDEX
PDS84             JPQ-3   000CE000    1   251 8p 032240-12 JPA SYSLIB PML CDEX
```

Now, I've read into storage the 2nd instance of the same load module, and I've written down that the addresses of these two instances are 000CE000 and 00105000.

list lkedmap pds84

```
ADDRESS    LENGTH   COMMENTS                  MODULE.CLASS/CSECT/LABEL
000CE000 (00036AA8) EP (MAJOR)                PDS84
```

I now know how long the load module is. I've also noted that the load module's entry address coincides with its load point, and I've written down that its length is 36AA8 bytes.

equate first ce000 36aa8
equate second 105000 36aa8
I've now assign equates to represent the starting addresses (and length) of the two instances.

map pds84+0
```
READING (VIA BSAM) THE MODULE MAP FOR PDS84 FROM PDS CSW.LINKLIB(PDS84)
READING (FROM SYM RECORDS) THE CSECT MAP FOR PDS84.PDSMAIN
```

list lkedmap pds84
```
ADDRESS    LENGTH   COMMENTS                  MODULE.CLASS/CSECT/LABEL
000CE000 (00036AA8) EP (MAJOR)                PDS84
      +0 (00002734) (MAPPED)                  PDS84.PDSMAIN.
     +44                                           .KLEAR.
    +4FE                                           .NEWCMD.
    +91A                                           .CMDSCAN4.
    +EE0                                           .CONVDATE.
   +110E                                           .LOGDATA.
   +1AC0                                           .OPENEXIT.
   +1CB8                                           .RETURN.
   +2738 (0000068B)                            PDS84.PDSINIT.
   +2DC8 (0000015E)                            PDS84.ATTNEXIT.
```

list lked second
```
ADDRESS    LENGTH   COMMENTS                  MODULE.CLASS/CSECT/LABEL
00105000 (00036AA8) EP (MAJOR)                PDS84
```

Just to keep things interesting, I've mapped (in the normal way) the first instance of PDS84 and one of its csects (PDSMAIN). Notice that only the module's first instance has been mapped, not the second.

dmap pds84.
```
READING (VIA BSAM) (FROM ESD INFORMATION) THE DSECT MAP FOR PDS84 FROM PDS
CSW.LINKLIB(PDS84)
```

Now I've read the module's module map AGAIN, but this time as if it were a dsect map. (Note, the command's trailing dot is important here.)

list maps
```
THE FOLLOWING MODULES AND CSECTS ARE MAPPED
  LKED: PDS84 -------------------------- 000CE000
 CSECT:   "  .PDSMAIN ------------------ 000CE000

THE FOLLOWING DSECTS ARE MAPPED
PDS84 ------------ INACTIVE
```

PDS84's module map now exists twice, once as a module map, and again as a dsect map.

dmap pds84#2 pds84;l maps

```
THE FOLLOWING MODULES AND CSECTS ARE MAPPED
  LKED: PDS84 ------------------------- 000CE000
 CSECT:   "  .PDSMAIN ----------------- 000CE000

THE FOLLOWING DSECTS ARE MAPPED
PDS84#2 ---------- INACTIVE
PDS84 ------------ INACTIVE
```

Now I've created a clone of the dsect map. It has a unique name.

using pds84#2 second;l maps

```
THE FOLLOWING MODULES AND CSECTS ARE MAPPED
  LKED: PDS84 ------------------------- 000CE000
 CSECT:   "  .PDSMAIN ----------------- 000CE000

THE FOLLOWING DSECTS ARE MAPPED
PDS84#2 ---------- 00105000
PDS84 ------------ INACTIVE
```

Now I've assigned the clone to represent the 2nd instance of the PDS84 load module.

format pds84#2.pdsmain

```
00105000 (A.S.DBCOLE6) --- PDS84+0, PDS84#2+0, SECOND+0, PRIVATE+104000
      +0                   PDS84#2  EQU   * PDSMAIN SECOND
      +0  47F0 F024        PDS84    B     24(,R15)
      +4  1FD7                      SLR   R13,R7
      +6           C4E2D4C1 C9D54040 E5F84BF4 404040F0  *DSMAIN  V8.4   0*
     +16           F261F1F7 61F9F540 40F1F94B F1F690EC  *2/17/95  19.16..*
     +26           D00C189F 41B00800 41A9B800 41BAB800  *}........z......*
     +36           41000000 5880B4C8 07F81B00 0000      *.......H.8....*
     +44  KLEAR    412094FE 5880B4CC 07F8D200 72867287  *..m......8K..f.g*
     +54           58F0B4F4 D70371DC 71DC05EF 47F0AC9C  *.0.4P........0..*
```

I can now display the location of the PDSMAIN csect (for example) within the 2nd instance of PDS84. **But** I've not yet mapped this csect.

format pds84.pdsmain

```
000CE000 (A.S.DBCOLE6) --- PDS84.PDSMAIN+0, FIRST+0, PDS84+0, PRIVATE+CD000
      +0                   PDSMAIN  EQU   * FIRST
      +0  47F0 F024        PDS84    B     24(,R15)
      +4           1F                31                  *.*
      +5  PDSCSECT D7C4E2D4 C1C9D540 40                  *PDSMAIN  *
      +E  PDSRELNO E5F84BF4 404040                       *V8.4   *
     +15  PDSDATE  F0F261F1 F761F9F5 4040                *02/17/95  *
     +1F  PDSTIME  F1F94BF1 F6                          *19.16*
     +24                   PDSCEND  EQU   *
     +24  90EC D00C        PDSSTM   STM   R14,R12,C(R13)
```

The same csect in the first instance of PDS84 has already been mapped.

NOTICE: The above two commands:
(FORMAT PDS84.PDSMAIN)
(FORMAT PDS84#2.PDSMAIN)
appear to have the same syntax, but they actually are very **DIFFERENT!**
- The address expression in the first command is "load module name dot csect name".
- The address expression in the second is "dsect name dot field name".

As far as the 2nd map is concerned (the PDS84#2 **dsect** map), **PDSMAIN** is **not** a csect name. It is just the name of a field within the dsect (just like PSATOLD is just the name of a field within the PSA dsect map).

To avoid confusion, it is very important to try to keep these points in mind when using module maps that have been loaded as dsects:

- Address expressions that reference load modules, csects and fields within csects can have up to three-parts: module.csect.field.

- Address expressions that reference dsects and fields within dsects can have up to only two-parts: dsect.field.

- Binder maps loaded as dsects are dsect maps (not Binder maps).

So for example:
- **FORMAT PDS84.PDSMAIN.** (notice the trailing dot) is a valid three-part name because PDS84 is the name of a load module.
- **FORMAT PDS84#2.PDSMAIN.** is an **invalid** three-part name because PDS84#2 is the name of a dsect.
- (Note, the trailing dot designates a null 3rd part name in the two address expressions above: Allowed in the first case, disallowed in the second.)

For more information, see **HELP ADDRESSING SYNTAX BASETERM.**

dmap pds84.pdsmain
READING (VIA BSAM) (FROM SYM RECORDS) THE DSECT MAP FOR PDSMAIN FROM PDS CSW.LINKLIB(PDS84)

list maps
THE FOLLOWING MODULES AND CSECTS ARE MAPPED
LKED: PDS84 ------------------------- 000CE000
CSECT: " .PDSMAIN ----------------- 000CE000

THE FOLLOWING DSECTS ARE MAPPED
PDSMAIN ---------- INACTIVE
PDS84#2 ---------- 00105000
PDS84 ------------ INACTIVE

Now, I've loaded PDSMAIN's csect map as if it were a dsect.

using pdsmain pds84#2.pdsmain float

list maps

```
THE FOLLOWING MODULES AND CSECTS ARE MAPPED
  LKED: PDS84 ------------------------- 000CE000
 CSECT:   "  .PDSMAIN ----------------- 000CE000

THE FOLLOWING DSECTS ARE MAPPED
PDSMAIN ---------- PDS84#2.PDSMAIN - 00105000
PDS84#2 ---------- 00105000
PDS84 ------------ INACTIVE
```

Now I've assign the PDSMAIN dsect map to represent the storage pointed to by the PDSMAIN field within the PDS84#2 dsect map.

The symbol "PDSMAIN" now occurs in **three** contexts:

- **1st:** It is the name of a csect within the PDS84 module map. This instance (and labels within it) can be reached by any of the following expressions:
  PDS84.PDSMAIN
  PDS84.PDSMAIN.
  PDS84.PDSMAIN.label

  And once a suitable **SET QUALIFIER** command has been issued, it (and its contained labels) can be reached by these expressions as well:
  .PDSMAIN.
  .PDSMAIN.label
  .label

- **2nd:** PDSMAIN also is the name of a field within the PDS84#2 dsect (module) map. This instance can be reached by any of the following expressions:
  PDS84#2.PDSMAIN
  .PDSMAIN
  (Note that this instance of PDSMAIN is a field label. It does NOT contain other labels within it.)

- **3rd:** Finally, PDSMAIN also is the name of a dsect map. This instance (and included labels) can be reached by the following expressions:
  PDSMAIN
  PDSMAIN.
  PDSMAIN.label
  .label

  (Note that the **SET QUALIFIER** command is not relevant for the accessibility of labels within dsects.)

- Notice that the previously issued **USING** command **(U PDSMAIN PDS84#2.PDSMAIN FLOAT)** has caused the last two instances of PDSMAIN to represent the same location in storage.

When using partially qualified references to the PDSMAIN name, the choice of **dots** is extremely important. For more information about this, see **HELP ADDRESSING PARSERS ASM RESOLUTION.**

Here are a Few Examples:

format pdsmain

```
00105000 (A.S.DBCOLE6) --- PDSMAIN+0, PDS84+0, PDS84#2+0, SECOND+0,
00105000 --- PRIVATE+104000
      +0                  PDSMAIN  EQU   * PDS84#2 SECOND
      +0  47F0 F024       PDS84    B     24(,R15)
      +4           1F                31                *.*
      +5  PDSCSECT D7C4E2D4 C1C9D540 40                *PDSMAIN  *
      +E  PDSRELNO E5F84BF4 404040                     *V8.4   *
     +15  PDSDATE  F0F261F1 F761F9F5 4040              *02/17/95  *
     +1F  PDSTIME  F1F94BF1 F6                         *19.16*
     +24                 PDSCEND  EQU   *
     +24  90EC D00C      PDSSTM   STM   R14,R12,C(R13)
```

This references the instance of PDSMAIN that is the name of a dsect map. It resolves to the location of the PDSMAIN csect in the second instance of PDS84.

format .pdsmain.

```
F .PDSMAIN.
            *
DBC035E DEFAULT MODULE NAME NOT DEFINED
```

set qualifier pds84.pdsmain

```
DEFAULT MODULE NAME IS PDS84
DEFAULT  CSECT NAME IS PDSMAIN
DEFAULT DSECT MAPS MODULE IS PDS84
```

format .pdsmain.

```
000CE000 (A.S.DBCOLE6) --- PDS84.PDSMAIN+0, FIRST+0, PDS84+0, PRIVATE+CD00
     .+0                 PDSMAIN  EQU   * FIRST
     .+0  47F0 F024      PDS84    B     24(,R15)
     .+4           1F                31                *.*
     .+5  PDSCSECT D7C4E2D4 C1C9D540 40                *PDSMAIN  *
     .+E  PDSRELNO E5F84BF4 404040                     *V8.4   *
    .+15  PDSDATE  F0F261F1 F761F9F5 4040              *02/17/95  *
    .+1F  PDSTIME  F1F94BF1 F6                         *19.16*
    .+24                PDSCEND  EQU   *
    .+24  90EC D00C     PDSSTM   STM   R14,R12,C(R13)
```

The first **FORMAT** command above attempts to refer to the instance of PDSMAIN that is the name of a csect within the PDS84 module map. However, this fails because a suitable **SET QUALIFIER** command has not yet been issued.

Once the **SET QUALIFIER** command has been issued, the **FORMAT** command succeeds and so it resolves to the location of the PDSMAIN csect in the first instance of PDS84.

Displays of both instances of PDS84 are now suitably formatted by appropriate Binder maps and csect maps.

Help DEbugging TIps S0c4

Sometimes it will happen that a program will "run fine" outside of Z/XDC but will fail (usually with an 0C4) when executed within Z/XDC. **OR VICE VERSA!** The program will run fine when executing within Z/XDC, but will fail when run by itself. Here are some common causes of these frustrating inconsistencies.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

ESTAES - Perhaps the most common cause of mystery 0C4s (and other ABENDs) is the presence of user program ESTAEs that receive control ahead of Z/XDC.

BADPOINTERS - Sometimes, a program will use a pointer that hasn't been properly initialized.

INITIALREGS - Z/XDC does **not** initialize R2-R12 to zeros.

SYSTEMCB - The presence of Z/XDC has unavoidable effects on certain system control blocks.

Help DEbugging TIps S0c4 Estaes

Perhaps the most common cause of mystery 0C4s (and other ABENDs) is the presence of user program ESTAEs that receive control ahead of Z/XDC before Z/XDC has a chance to see the ABEND. Consider the following hypothetical scenario:

- A debugging session is started using the XDCCALL command (or any of its aliases) to setup and start the session. (See HELP XDCCALL.) XDCCALL setups up Z/XDC as an ESTAI for your program, and then passes control to Z/XDC so that you can load maps, set breakpoints, etc. Eventually, you will type GO thereby allowing your program to start executing.

- During its initialization, your program issues an ESTAE macro (for example) to to create a recovery routine for handling ABENDs. This recovery routine will be newer than the ESTAI that is Z/XDC, and so when an ABEND occurs, the user program's ESTAE will be given control ahead of Z/XDC.

- Subsequently, a breakpoint is reached. Remember, breakpoints are simply X'00' opcodes, and so as far as everybody other than Z/XDC is concerned, a breakpoint is nothing more than an s0C1 ABEND. (See HELP BREAKPOINTS.)

- The system sees the s0C1, and sees that the newest recovery routine is the user program's ESTAE, and so passes control to that ESTAE.

- The user ESTAE typically doesn't know anything about breakpoints, and so all it sees is an s0C1 that it most certainly doesn't expect. Often in such a situation, the ESTAE will decide that the main program must terminate, and so it will perform some initial shutdown processing such as closing datasets, **FREEMAIN'ing data areas,** setting footprints, scheduling dumps, etc. It then will percolate so

that the system can produce the dump and terminate the program.

- But there is an older recovery routine, the ESTAI for Z/XDC. So instead of terminating the program, the system passes control to this next ESTAI, and so Z/XDC comes alive and displays its screens. Unfortunately, in this situation, it is **not obvious** that the user program's ESTAE has already executed and has already performed initial shutdown processing.

- So the user will proceed to issue a variety of commands, typically ending with a TRACE or GO/GOT/GOX command. This allows the user program to do something that it had no way of anticipating: Continue execution even after initial shutdown processing has already occurred!

- Eventually, the user program will try to do something in contradiction to the initial shutdown processing and so will fail, maybe with an 0C4, maybe with almost any symptom conceivable.

Probably the best tool for detecting this problem is the **LIST ESTAES** command. This command produces a report showing the status of all recovery routines that are associated with the current task. SCB#1 will be listed first and describes the task's oldest recovery routine. The SCB described by the report as being "ACTIVE", will always be where Z/XDC is running.

When Z/XDC is properly set up, it will be running as the **second newest** recovery routine. I.e. the second to last displayed SCB will be the one that is described as being ACTIVE. (The newest recovery routine will always be Z/XDC's own internal ESTAE, it will always be PENDING, its exit address will always be deep within the XDC load module, and it can always be ignored since it will be canceled whenever a TRACE or GO/GOT/GOX command is used to return control to the user program.)

If Z/XDC is not running as the second newest recovery routine then most likely the second newest recovery routine will be a user program routine, and it will be marked as "PERCOLATED". If this is the case, then user ESTAE interference has occurred, and the program environment **cannot be trusted!**

To cope with and prevent the problems arising from an interfering ESTAE, do the following:

- First, you will need to know what parts of your programs are executed under which tasks. You will also need to know which parts of your programs are executed under special RBs (Request Blocks). Such routines might be:
  - DCB Open Exits
  - Timer exits
  - ESTAE exits
  - Attention exits
  - etc.

For more information, see HELP EXECUTIONLEVELS.

- Then you will need to know which ESTAEs your program uses to protect which parts of your programs.

- Then when you are about to set a breakpoint into a part of a program that will have a recovery environment that is different from the current recovery environment, then use the HOOK or HDEFERRED command instead of the breakpointing command that you would normally use.

Unlike breakpoints, hooks are SVC instructions that give a special support routine

the opportunity to create the proper debugging environment before passing control to Z/XDC. See HELP HOOKS for more information.

For more information about dealing with interfering ESTAEs, see HELP DEBUGGING ESTAES.

Help DEbugging TIps S0c4 Badpointers

Sometimes a program will load and use an address pointer that hasn't been properly initialized. When this bug is present, then nearly anything can happen. The most common result though will usually be an 0C4 ABEND.

When a bad pointer is in use, the stability of the program becomes highly dependent upon what I call "the shape of storage". By this I am referring to the particular locations of allocated and unallocated storage. Whether or not a bad pointer will cause an 0C4 simply depends upon whether that pointer happens to point to allocated storage or unallocated storage.

Although the shape of storage tends to be random, at the same time it often is repeatable. In other words, if a program and its relevant data does not change substantially, then if you run that program over and over again, the shape of storage created by that program typically will not change. Consequently, if the use of a bad pointer happens not to cause an 0C4, then it probably will continue not causing an 0C4 on subsequent executions.

The presence or absence of Z/XDC in your program's environment, however, has a tremendous effect upon the shape of storage, and so Z/XDC's presence or absence is likely to affect whether or not a bad pointer will cause an 0C4.

From time to time I will hear the complaint, "My program was running fine until I tried running it under Z/XDC. Then 0C4s started happening." A bad pointer may be the reason. (For other possible reasons, see HELP DEBUGGING TIPS S0C4.)

On the other hand, I sometimes even hear the **opposite complaint:** "My program was running fine under Z/XDC and didn't fail until I tried running it **without** Z/XDC!" When this occurs, it is almost certainly because of a bad pointer.

Help DEbugging TIps S0c4 Initialregs

When the operating system first passes control to a program, it initializes R13, R14, R15, and R1 to various values, and all other registers to zeros. Z/XDC mimics the system with respect to R13, R14, R15, and R1, but does **NOT** clear the other registers to zeros! (This is a feature, not a bug, so please don't bother complaining about it. Sorry.) If your program depends upon a zeroed value in any of these registers, then the results will be unpredictable.

Help DEbugging TIps S0c4 Systemcb

The presence of Z/XDC in your program's environment has some unavoidable effects upon

various system control blocks and structures. If your program should happen to have dependencies upon these structures, then unpredictable results may occur. Some of Z/XDC's effects are as follows.

JOBSTEP TASK
If you use XDCCALL (or any of its aliases) to start your debugging session, then XDCCALL, **not** your program, will be running as the jobstep task. Your program will be running as a subtask of XDCCALL.

R14
Normally, when the system passes control to a program, R14 will point to an SVC 3 instruction located in the system's CVT. But when XDCCALL is used to start the debugging session, R14 instead points to a special dead trap located within a Z/XDC owned data area. Accordingly, your program should not issue an SVC 3 instruction to terminate. Instead, it is better if it uses R14 to terminate.

LOAD LISt ELEMENT (LLE)
If you use XDCCALL (or any of its aliases) to start your debugging session, then your main program will be brought into storage via a LOAD macro, not an ATTACH macro. This means that a Load List Element (LLE) will exist for your program. (It's hard for me to imagine why this might matter, but you never know.)

Help ADdressing

Z/XDC command operands that refer to virtual storage are called "address expressions". A large number of Z/XDC commands use address expressions as operands. The FORMAT command, for example, expects its first operand to be an address expression. (The FORMAT command generates a formatted display of virtual storage.) Some other commands that use address expressions are:

- **AT** - Sets persistent breakpoints.
- **ATX** - Sets very persistent breakpoints.
- **COPY** - Copies data from one storage location or register to another.
- **DISPLAY** - Displays virtual storage in a basic raw hex-text (dump like) format.
- **EQUATE** - Creates individual symbols assigned to storage.
- **FIND** - Searches storage for a given hex or character string.
- **FREEMAIN** - FREEMAIN's a given area of virtual storage.
- **GO** - Resumes user program execution at a given address.
- **HOOK** - Sets hooks to activate Z/XDC debugging sessions in other programs.
- **LIST VAR** - Displays a list of HLL variables for a program being tested with the c/XDC feature.
- **MAP** - Reads a module map and a csect map for a load module and control section at a given address.
- **OFF** - Removes breakpoints.
- **POST** - Posts an ECB.
- **SHOW** - Displays a list of storage locations.
- **TRAP** - Sets transient breakpoints.
- **USING** - Assigns dsect maps to storage.
- **VERIFY** - Verifies the contents of storage, a register or a PSW.
- **ZAP** - Changes the contents of storage, a register or a PSW.

Types of Storage References

Virtual storage can be referenced either symbolically, absolutely, or implicitly. An absolute reference is a hexadecimal number. A symbolic reference is a name of an object (load module, dsect map, field name, variable name, statement#, equate name, etc.) that resides at a specific storage location. An implicit reference is a re-reference to a storage location that Z/XDC has just previously displayed or zapped.

An Address Expression's Basic Parts

Address expressions have three distinct parts. All of these three parts are optional under various circumstances.

- The **base term** defines a starting point for the address expression. A base term establishes an address expression's initial "location calculated so far". This may be any of the following:
  - A register
  - A PSW
  - A raw hexadecimal address
  - Certain built-in functions
  - The name of a:
    - Load module
    - Program object
    - Csect
    - Dsect
    - Field within a C/Dsect
    - Built-in or automatic equate
    - User-created equate
    - High Level Language variable reference
    - High Level Language statement number

  (In the case of PSWs and registers, the reference **must** be followed immediately by an "indirect operator", indicating that the register or PSW is being used as a pointer to storage.)

- **Offset terms** start with plus or minus signs and compute values to be added to or subtracted from the location calculated so far.

- **Indirect operators** may occur only between terms. They cause a new address to be loaded from the place pointed to by the old location calculated so far (from a register, from a PSW, or from storage). This new location replaces the old location calculated so far.

Built-in Functions

Z/XDC also has several built-in functions that can be used in address expressions. These functions do such things as:

- Change the address space or dataspace to which the expression is targeted,
- Extract values from storage for use in the expression's further computation,
- Perform certain arithmetic or logical transforms (truncating, shifting, etc.) on the address calculated so far,
- Perform indirect addressing operations that are more complex than those performed by the basic **%** **?** and **!** operators.

Built-in functions generally have operands that may, themselves, be address expressions. So Z/XDC's address expression resolution process can be a recursive affair.

Parsers, the Parse Order and Tagged Address Expressions

In its support of multiple High Level Languages, Z/XDC has to cope with the fact that the syntaxes for variables are different from one language to the next. Consequently, Z/XDC contains several **Language Parsers,** one for each supported programming language.

Z/XDC provides controls by which you can manage (1) which parsers Z/XDC calls, (2) the order in which they are called, and (3) which parser produces the error message should all parsers fail. For more information, see HELP ADDRESSING PARSERS.

Basic Examples

Below are several examples of some simple address expressions. (These examples assume that your program is **not** in cross memory mode.) In these examples, all addresses are relative to the Target Address Space which, in these examples, is the Home Space.

FORMAT 10	- Displays the contents of the CVTPTR field in low storage.
FORMAT 10%	- Displays the start of the System's CVT.
FORMAT 21C%	- Displays the current program's TCB.
FORMAT 21C%+70%	- Displays the current program's first register save area.
FORMAT 21C%+70%+8?	- Displays the current program's second register save area.
FORMAT TCB.TCBFSA%+8?	- Also displays the current program's second register save area.
FORMAT R14?	- Displays the location pointed to by register R14 using the 31 lo-order bits of R14.
FORMAT R14%	- Displays the location pointed to by register R14 using only the 24 lo-order bits of R14 (i.e. ignoring the hi-order byte of R14).
FORMAT R14!	- Displays the location pointed to by register RW14 using all 64 bits or RW14.
FORMAT RW14!	- Ditto. (See HELP COMMANDS LIST GENERALREGISTERS INDIVIDUAL for a discussion of register names.)
FORMAT PSW?	- Displays the next instruction to be executed by the current program.
FORMAT +0	- Redisplays the same location most recently displayed.
ZAP +0=4700	- Zaps two bytes of data into the location starting at the most recently displayed (or verified) location.
FORMAT .+2BE	- Displays the location at offset X'2BE' into the current default csect. - The dot refers to the zero point for the default csect. - The default csect is established by the SET QUALIFIER command.
FORMAT AR5?+10	- Displays a location that is 16 bytes (decimal) past the location pointed to by **general** register R5 in the data space or address space designated by **access** register AR5.
FORMAT AR5?+16N	- Ditto.

```
FORMAT R5?~ALET(AR5)+10 - Ditto.
FORMAT R6?~ALET(AR5)    - Displays a location that is pointed to by general register
                          R6 in the address/data space designated by access register
                          AR5.
FORMAT R6?~ALET(R1%)    - Displays a location that is pointed to by general register
                          R6 in the address/data space designated by an ALET located
                          in virtual storage pointed to by general register R1.
```

The following examples apply to programs running in Cross Memory (XMS) Mode. (See HELP DEBUGGING PCROUTINES for more information.):

```
FORMAT R5?              - Displays, in the retry level Primary Address Space, the
                          location pointed to by the retry level instance of general
                          register R5.
FORMAT ER5?             - Displays, in the error level Primary Address Space, the
                          location pointed to by the error level instance of general
                          register R5.
FORMAT R5?~ALET(1)      - Displays, in the retry level Secondary Address Space, the
                          location pointed to by the retry level instance of general
                          register R5.
FORMAT R5?~ALET(2)      - Displays, in the Home Address Space, the location pointed
                          to by the retry level instance of general register R5.
                          (The Home Address Space is the same space in both the
                          error level and retry level execution environments.
```

(For a discussion of the **retry level** environment vs. the **error level** environment, see HELP EXECUTIONLEVELS.)

In order to reduce the need for cumbersome punctuation, Z/XDC permits the use of syntactically ambiguous symbolic references. Such references are resolved according to a predefined structure of "preferences".

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

PARSERS - Language Parsers and their management.
NUMERIC - Numeric storage references.
DOTPLUS - Storage references to a default csect.
IMPLICIT - Implicit storage references.
TARGETSPACES - Information about how Z/XDC decides upon what address space to display.
FUNCTIONS - Built-in addressing functions.
SYNTAX - Structure of address expressions.
EXAMPLES - More examples.

Help ADdressing Parsers

In its support of multiple High Level Languages, Z/XDC has to cope with the fact that the syntaxes for variables are different from one language to the next. Consequently, Z/XDC contains several **Language Parsers,** one for each supported programming language.

So when parsing the **base term** of an address expression, Z/XDC first calls one parser and then another until all parsers have been called.

When one parser finally succeeds in parsing the base term without error, then the result of that parse (basically a location in storage) is accepted, and common code is then executed to parse the rest of the address expression.

When **all** parsers fail, the error message generated by the last parser called is displayed, and the command is aborted. (Note, you will probably want to set it up so that the last parser called is the same as the first parser so that the generated error message makes sense. This is permissible. See HELP ADDRESSING PARSERS PARSEORDER for more information.)

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

PARSEORDER - Defining the order in which selected parsers are called.
TAGS - Designating a particular parser to be called for the current address expression.
ASM - The syntax rules used by the Assembler Parser.
CEE - The syntax rules used by the C Language Parser.

Help ADdressing Parsers Parseorder

In order to support multiple programming languages, Z/XDC needs to be able to parse variable names according to language specific syntax rules. So Z/XDC supports multiple **Language Parsers** (one for each supported programming language) for parsing variable names.

In order to facilitate debugging programmed systems written in multiple languages, Z/XDC uses the idea of a **Parsing Order** which operates according to the following rules:

- (1) When Z/XDC encounters a possible variable name in an address expression, it will start calling language specific parsers to parse the name.

- (2) If one parser fails, it will call another parser until either the name is successfully parsed or all parsers fail.

- (3) When a parser succeeds, it returns the variable’s location in storage for use in resolving the rest of the address expression. All remaining parsers are

bypassed.

- (4) If all parsers fail, then the current command is aborted, and the error message produced by the last parser is displayed. (It is permissible that said last parser be made the same as the first so that the error message will make sense in its context.)

You have full control over the parsing order. You can display it, and you can make your changes permanent by saving it into your Z/XDC Profile.

- To display the current Parse Order, use the **LIST PARSEORDER** command.
- To change the Parse Order, use the **SET PARSEORDER** command.
- To make your changes permanent, use the **PROFILE SAVE** command.
- To make a one-time override of the current Parse Order, use **tagged** address expressions. (See the next following topic for details.)

Note, you can also display and change the Parse Order via the Profile Menuing System. See HELP PROFILES MENU READZAPPARSE PARSEORDER for more information.

Help ADdressing Parsers Tags

In order to support multiple programming languages, Z/XDC supports multiple language specific parsers which it calls in a user specified Parsing Order when it needs to parse a variable name. There may, however, be times when you would want to override that order and force Z/XDC to call a specific parser for a particular address expression. You can do that by **tagging** the address expression with the name of the desired parser.

Tagging is very simple. Just precede the address expression with the **nickname** of the parser you want, followed by a colon (:). Examples:

f asm:.tcbrbp?
This command parses the "tcbrbp" fieldname according Assembler language parsing rules (regardless of the current Parsing Order).

f cee:structure.arrayname
This command parses the "structure.arrayname" variable according C language parsing rules (regardless of the current Parsing Order).

Available Parsers

Currently, there are two parsers available. Their nicknames are:

ASM for Assembler
CEE for XL C/C++ and Metal C

Help ADdressing Parsers Asm

The Assembler Language Parser parses variable names according to the Syntax rules of the High Level Assembler language. Z/XDC may or may not use this parser according to:

- Whether or not it is listed in the currently active Parse Order (displayed/set by the LIST/SET PARSEORDER commands).
- If its use is explicitly forced by an address expression tag (ASM:---)

Z/XDC calls language specific parsers when parsing variable names. If Z/XDC calls this parser, then it will either succeed and return the storage location of the resolved variable, or it will fail and return some variation of a Syntax Error message.

The address expression's resolved location can then be adjusted as desired by the balance of the address expression in which the variable reference occurs.

Referencing Storage Symbolically

A symbolic storage reference is a reference that uses the name of an object located in storage. Such symbolic names may be constructed from either one, two, or three name elements (or "qualifiers") joined to each other by periods (e.g., "DSECTNM.LABEL").

Basically, there are eight kinds of symbolic names. Their forms are:

```
modulename.csectname.labelname
modulename.dsectname
dsectname.fieldname
equatename
hookname
breakpointname
registername
PSWname
```

Load module names, csect names, dsect names, equate names, hook names, and breakpoint names generally have the following syntax:

- Each character in the name element may be either numeric (except the first) or alphabetic or national ($ # or @) or an underscore (_).
- In addition, any character in a name element may be an open brace ("{" i.e. X'C0'). This is because the names of some SVC load modules end with an open brace.
- For dsect names, csect names, entry point names, ESD names, and C program label names, the case of the name that you type must match the name's actual case **only when** there exists multiple instances of the name that differ only by case. When only one such instance of a name exists, then case matching is never necessary. (See HELP ADDRESSING PARSERS ASM MIXEDCASE fore more information.)
- For load module names, Assembler instruction names and field names, the case of the name that you type never needs to match the name's actual case since for these categories, multiple names differing only by case can never happen.

OR Label and field names may instead follow these extended syntax rules:

- The name may be from 1 to 64 characters long.
- Any character (*including* the first character) may be a digit, an alphabetic, a national (# $ and @), a dash ("-"), or an underscore (_).
- The name must include at least one alphabetic character.

OR It is possible for the Assembler (for example) to construct Binder-time csect names and entry point names that violate syntax rules entirely! For this case, Z/XDC allows you to provide label names enclosed within tic marks (').

There are some rare circumstances under which it is possible for a csect map or a dsect map to contain multiple occurrences of a given label that are identical, including case. When this happens, Z/XDC will display the names with subscripts, and it will require you to use subscripts when referencing a specific instance of the name. Example: FORMAT MYDSECT.MYDUP(2) displays the second occurrence of the MYDUP label within the MYDSECT dsect.

When resolving an address expression, Z/XDC usually will use a case-insensitive method for looking up most symbolic names (equate names, load module names, Assembler statement/field names, and the like). The exception occurs for labels that occur within maps that contain case-sensitive labels. When a given label occurs two or more times and the occurrences differ only by case, then a case-sensitive match is required.

There is a special case where a name element can be an "extent number". This is for use with "multi-extent" load modules, ie. load modules, such as "scatter loaded" load modules or RMODE(SPLIT) program objects, that have been loaded into two or more separate areas ("extents") of storage.

An extent number may be from 3 to 5 characters long. The first two characters must be "X#". The remaining characters must constitute a decimal number indicating the desired extent. A load module's first (and usually only) extent is "X#1". Example: "IGC0001I.X#1" refers to the **physical start** of the first (and only) extent of the load module containing the System's OPEN SVC routine. For more information, see HELP ADDRESSING PARSERS ASM EXTENTS.

Note that a load module name, given without qualification, refers to the module's **entry address,** which may or may not be the same as the start of the module's first extent (also known as the module's "load point").

In order to refer to a module's physical start (guaranteed), qualify the module's name with **".X#1".** This means "the start of the module's first extent". Examples:

FORMAT XDCTFS
FORMAT XDCTFS.
FORMAT XDCTFS+0
These all refer to the entry point of the load module named XDCTFS.

FORMAT XDCTFS.X#1
This refers to the XDCTFS's load point.

Each of the multi-element symbolic names can be given with various of their elements omitted (e.g., ".labelname" or ".csectname."). When this is done, certain defaults are used.

Examples:

```
IEANUC01.SVCTABLE   - Refers to the system's SVC table.
TCB.TCBFSA          - Refers to the register save area anchor field in the TCB.
IGC0002{            - Refers to the entry point of the system's CLOSE SVC routine
                      (SVC 20).
IGC0002{.X#1        - Refers to the physical start of the load module that contains
                      the System's CLOSE SVC routine.
.1ST-TOTAL-LESS-2ND - Refers to a data field named "1st-total-less-2nd" within the
                      current default csect.
```

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

- **MIXEDCASE** - Which symbols are case-sensitive and which are not.
- **QUOTED** - Dealing with names that violate syntax.
- **RESOLUTION** - Using partially qualified names.
- **DUPLICATES** - Using subscripts to resolve duplicate names.
- **EXTENTS** - Multi-extent load modules.
- **LONGNAMES** - Objects with names longer than 8 characters.
- **ASMEQUS** - Dealing with Assembler equates, especially **non-relocatable** EQUs.
- **JPALPAPLPA** - Areas of storage containing load modules: JPA, LPA, MLPA, FLPA, DLPA, PLPA.
- **MODULES** - Referring to structures within load modules.
- **DSECTS** - Referring to dsects and their fields.
- **EQUATES** - Referring to Z/XDC defined equate names.
- **REGISTERS** - Referring to various register sets.
- **PSWS** - Referring to various PSWs.

Help ADdressing Parsers Asm MIxedcase

Z/XDC supports treating a symbol with sensitivity to its case but only when necessary. With respect to maps and the labels that they contain, case-sensitivity applies as follows:

- **Maps of Assembler Control Sections and Data Areas:** The Assembler supports mixed-case names but only in a case-insensitive way. That is to say, labels that are identical except for case are considered by the Assembler to be the same label. Therefore, whenever the MAP or DMAP command is used to build a map of an Assembler csect or dsect, Z/XDC flags that map as being case-insensitive.

- **Maps of Load Modules and Program Objects:** The Binder fully supports case-sensitivity. That is to say, labels that are identical except for case are considered by the Binder to be different labels. So when the MAP or DMAP command is used to build a module/object's module map, Z/XDC examines all section names, class names, label names, etc. to see if any of them contain lowercase letters. If any do, then the map is flagged as being case-sensitive; otherwise, it is flagged as being case-insensitive.

- **Source Image Maps of C Program Code:** The XL C compiler supports mixed-case labels. Therefore, when the **MAP** command builds source image maps for such programs, they are flagged as being case-sensitive.

When a map is case-sensitive, then references to labels within that map may or may not have to exactly match the case of the label being referenced. Here are the rules:

- If a map contains multiple instances of a label that differ only by case, then:
  - Any reference to any of those instances must exactly match the case of the desired instance.
  - If a reference matches the text of a label but not any variation of its case, then the attempted match fails, but an error message (DBC033E) will be produced that lists up to three of the label's variations.

- If a map contains only one instance of a label, then attempts to reference that label do not have to exactly match the case of that label. The attempt will always be allowed to succeed.

Note, if a map contains two or more labels that are duplicates both in text and in case, then subscripting has to be used to distinguish which of the duplicates you wish to match. For more information see HELP ADDRESSING PARSERS ASM DUPLICATES.

When a map is case-insensitive, then its labels can be referenced regardless of the case of the reference.

Z/XDC treats all other kinds of names as being case-insensitive. Specifically, the following names are always case-insensitive:
- Load module names
- Program object names
- Machine instruction names
- Assembler data field names
- Equate names
- Breakpoint names
- Hook names

Mixed casing is not supported for load module names and program object names because in z/OS, these names still must follow the standard rules for PDS member names.

Mixed casing is not supported for machine instruction statement names and Assembler data field names because the High Level Assembler always considers two symbols whose names differ only by case to be the same symbol. In other words, within the Assembler, a mixed-case variant of a symbol is the identically same symbol as its uppercase variant.

Examples:

Assume that **TESTLOAD** is a program object that contains the following mixed-case control section names:
    PayRoll
    DATAcsect
    DataCsect
Assume that there is no other instance of a csect having a case-variant of the name PayRoll. Then the following commands will have the following results:

FORMAT testload.payroll.firstpass
This command succeeds. It references two maps:

- A module map of the program object named TESTLOAD. This map has been flagged as being case-sensitive because it contains control sections having mixed-case names: PayRoll, DATAcsect and DataCsect.

- A csect map of the control section named PayRoll. This map has been flagged as being case-insensitive because the Assembler always treats all internal labels as being case-insensitive.

The FORMAT command's reference to **TESTLOAD** succeeds because load module names are always treated as being case-insensitive.

The FORMAT command's reference to **PayRoll** succeeds because the TESTLOAD map contains only one instance of a "payroll" name, so case matching is not required.

The FORMAT command's reference to **FIRSTPASS** succeeds because the Assembler does not distinguish names by their case.

FORMAT testload.datacsect.paydata
This command fails because **both** of the following are true:
- The name **datacsect** does not match the case of either instance of the datacsect name that occurs within the TESTLOAD module.
- The TESTLOAD module contains more than one occurrence of the datacsect name. So case matching is required to distinguish which instance is desired.

When this failure occurs, the following error messages are issued:

```
DBC033E CSECT/EXTERNAL NAME "datacsect" NOT FOUND IN THE PROGRAM
DBC033E OBJECT MAP NAMED "TESTLOAD". 2 LABELS WERE FOUND THAT
DBC033E WOULD HAVE MATCHED, BUT FOR CASE
DBC033E THOSE LABELS ARE:
DBC033E     DATAcsect
DBC033E     DataCsect
```

This makes it easy for you to correct your mistake.

Help ADdressing Parsers Asm Quoted

In recent years, it has become possible for programmers (using the Assembler's ALIAS instruction) to create csect names and entry point names that violate all rules of syntax. Such names may contain special characters blanks, embedded quotes, binary values, whatever...

In order to make it possible to reference such names in address expressions, Z/XDC now permits names to be given enclosed by tic marks (’). The following parsing rules apply:

- Precede and follow the name with a tic marks (’). These quotes are not a part of

the name, and so they will be stripped off by parsing.

- If a quoted name contains quotes as a part of the name, then those quotes need to be doubled up. (Example: 'don''t')

- Alphabetic characters within the name are treated as being case-sensitive when necessary and case-insensitive otherwise. Specifically:
  - When a module contains two or more instances of a quoted name that are identical except for case, then a given quoted name must match one of those instances exactly, including case.
  - Otherwise, a case match is not necessary.

  For more information, see HELP ADDRESSING PARSERS ASM MIXEDCASE.

- Everything else enclosed within matching quotes is ignored with respect to syntax.

- For compound names (names separated by dots), each element in the name may be (but does not necessarily have to be) enclosed with quotes. The entire compound name, however, must not be so enclosed. Examples:

  - These work:
    'modulename'.'csectname'.'labelname'
    modulename.'csectname'.labelname
    (etc.)

  - This doesn't work:
    'modulename.csectname.labelname'

Examples:

If you would like to experiment with quoted names, then assemble the following program:

```
BADNAMES ALIAS C'BadName:'
BADNAMES CSECT ,
         DC    C'BAD CSECT NAME'
         DC    XL(32-(*-BADNAMES))'00'

BADNTWOA ALIAS C'Bad'':Two'
         ENTRY BADNTWOA
BADNTWOA DC    C'FIRST BAD ENTRY NAME'
         DC    XL(32-(*-BADNTWOA))'00'

BADNTWOB ALIAS C'BAD'':Two'
         ENTRY BADNTWOB
BADNTWOB DC    C'SECOND BAD ENTRY NAME'
         DC    XL(32-(*-BADNTWOB))'00'

         END   ,
```

Assume that this program has been assembled and linked into a load module named **BNTESTS.** Then the following commands will have the following results.

map bntests
A module map is loaded for the BNTESTS load module.

s q bntests
s q bntests.'badname:'
Both of these commands accomplish the same thing:
- **BNTESTS** is made the default load module name.
- **BadName:** is made the default csect name.

format .
f bntests.'badname:'
f bntests.'BADNAME:'
f .'BADNAME:'.
All of these commands accomplish the same thing: They display the area of storage where the **BadName:** csect resides.

Note that in this case, it is not necessary for the address expression to exactly match the case of the csect's name.

format .'Bad'':Two'.
format .'BAD'':Two'.
format .'bad'':two'.
Each of these commands has a different result:

- The first FORMAT command exactly matches the entry point named **Bad':Two,** so the storage location containing the text string "FIRST BAD ENTRY NAME" is displayed.

- The second FORMAT command exactly matches the entry point named **BAD':Two,** so the storage location containing the text string "SECOND BAD ENTRY NAME" is displayed.

- The third format command does not exactly match anything. It matches the text of more than one label, so a case match then is mandatory. However, the given name does not match the case of either label, so the command is failed with error message DBC033E.

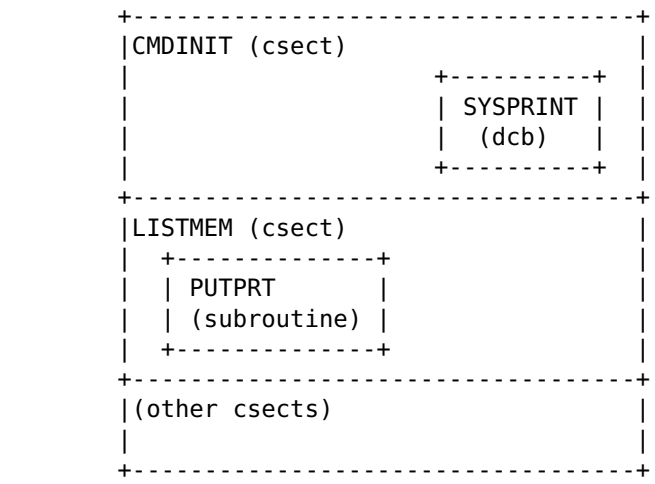
Help ADdressing Parsers Asm RESolution

Multi-element names can be given with various of the elements omitted. For example, if a given name starts with a period, then one or two leading elements have been omitted. If a given name is shorter than three elements, then one or two trailing elements may have been omitted (depending upon context). If a given name ends with a period, then trailing elements have, for sure, been omitted.

Example: Suppose that a load module named SYSCMD resides in storage and has the following characteristics:
- It contains at least two csects. One is named CMDINIT and the other is named

LISTMEM.
- LISTMEM contains a subroutine labeled PUTPRT.
- CMDINIT contains a Data Control Block named SYSPRINT.

Pictorially, SYSCMD might appear as follows:



Z/XDC has excellent tools for representing structures like this symbolically. In this example, the following commands can be used to build a symbolic representation of SYSCMD:

M SYSCMD.CMDINIT
M SYSCMD.LISTMEM
These commands cause Z/XDC to read csect maps for the CMDINIT and LISTMEM csects. In addition, the first command (being first) also causes Z/XDC to read a module map of the SYSCMD load module. Once these maps are read, the symbols that they contain can be used in other Z/XDC commands in address expressions.

Note: Csect maps can be read by Z/XDC only when either SYM records or ADATA records are available:
- SYM records are available when a program is both assembled and linkedited with PARM='TEST,...' specified for both the assembly step and the linkedit or Binder step.
- ADATA records are available when a program is assembled with PARM='ADATA,...' specified.

See HELP MAPS for more information.

S Q SYSCMD.LISTMEM
This command establishes SYSCMD as the "default" load module and LISTMEM as the "default" csect. This makes it easier to use LISTMEM's internal labels in address expressions: They don't have to be prefixed by "SYSCMD.LISTMEM.".

DM XDCMAPS.IHADCB
U IHADCB SYSCMD.CMDINIT.SYSPRINT
The DMAP command causes Z/XDC to read a dsect map named IHADCB (containing DCB field names). The USING command assigns that map to represent the fields of the SYSPRINT DCB located in the CMDINIT csect.

EQ IMHERE PSW? I F
The EQUATE command assigns a "floating" label named "IMHERE" to represent the user program's current execution location. Note, because IMHERE floats, it will

follow the progress of execution. (See HELP EQUATES for more information.)

Once the above commands have been successfully issued, the following sets of labels become available for use by Z/XDC commands:

- The names of all csects and common blocks within the SYSCMD load module.
- The names of all statements and fields within the CMDINIT and LISTMEM csects.
- The names of all fields within the SYSPRINT DCB control block.
- The "IMHERE" label representing SYSCMD's current execution location.

The following are examples of various ways that these symbols can be used in Z/XDC commands:

AT .
This command sets a breakpoint at the start of the LISTMEM csect. When a breakpoint is created, Z/XDC generates a label to represent the breakpoint. In this example, the generated label might be "AT00002B". Also note that "." is an address expression that refers to the start of the "default" csect established by the SET QUALIFIER command. In this example, the "AT ." command is equivalent to "AT SYSCMD.LISTMEM".

FORMAT SYSCMD.LISTMEM.PUTPRT
FORMAT SYSCMD.CMDINIT.SYSPRINT
These symbolic references are of the form "modulename.csectname.labelname". These two commands resolve to the addresses of the PUTPRT subroutine within the LISTMEM csect and the SYSPRINT DCB within the CMDINIT csect, respectively.

FORMAT .LISTMEM.PUTPRT
FORMAT .CMDINIT.SYSPRINT
This form is ".csectname.labelname". These two commands produce the same displays that would be produced by the prior example. These two commands resolve to the addresses of the PUTPRT subroutine and SYSPRINT DCB, respectively. The positions of the two dots in the name informs Z/XDC that it has been given the last two parts of a three part name. This form can only be used after a suitable SET QUALIFIER command has been issued (against **any** csect in the load module) making SYSCMD the default load module. Examples:

```
    S Q SYSCMD.LISTMEM
    S Q SYSCMD.CMDINIT
```
Or perhaps:
```
    S Q R15?
    S Q PSW?
```

FORMAT .LISTMEM.
FORMAT .CMDINIT.
This form is ".csectname.". These two commands resolve to the starting addresses of the LISTMEM csect and the CMDINIT csect, respectively. The position of the two dots in the name informs Z/XDC that it has been given the middle part of a three part name. Again this form can only be used after a suitable SET QUALIFIER command has been issued making SYSCMD the default load module.

FORMAT .PUTPRT
FORMAT .DCBLRECL
This form is either ".fieldname" (of a dsect) or ".labelname" (in a csect). It is the last part either of a 3-part "load-module.csect.label" name or of a 2-part "dsect.field" name. Therefore, when presented with this form, Z/XDC first searches the current default csect and then all active dsects until it finds the name. In these two examples, Z/XDC finds PUTPRT in the LISTMEM csect (because a SET QUALIFIER command was issued making LISTMEM the default csect), and it finds

DCBLRECL in the IHADCB dsect map (because DCBLRECL was not found in LISTMEM and it was found in the IHADCB map and that map is currently active, having been assigned a base address via the USING command).

FORMAT .SYSPRINT
Under the assumptions of this example, this command **fails.** This is because "SYSPRINT" is located in the CMDINIT csect and that is **not** the current default csect.

FORMAT .
FORMAT .+3E6

When a dot is given unassociated with any name, Z/XDC uses the zero point of the current default csect. In this case that would be the address of the LISTMEM csect within the SYSCMD load module. Note that offsets (such as "+3E6" in this example) can be added to "dot" just like they can to any address expression.

FORMAT +30
FORMAT +0
FORMAT -2

When an address expression starts with a operator (+ - % ? !), then something called the "Current Display Pointer" (CDP) is used as the starting point for the address computation. The CDP is an internal Z/XDC value that is set every time storage is displayed (or zapped by Z/XDC). For more information, see HELP ADDRESSING IMPLICIT.

NOTE: The "+0" address expression is particularly useful. In words, it means, "the location that I most recently displayed". If you have just displayed some location, ("FORMAT R15?" for example), than "+0" provides a very easy way to map that same location ("MAP +0"), zap that same location ("ZAP +0=47F0"), or whatever.

FORMAT SYSCMD.LISTMEM.
FORMAT SYSCMD.CMDINIT.
This form is "modulename.csectname.". The positions of the two dots in the name informs Z/XDC that it has been given the first two parts of a three part name. These resolve to the starting addresses of the LISTMEM csect and CMDINIT csect, respectively.

FORMAT SYSCMD.LISTMEM
FORMAT SYSCMD.CMDINIT
FORMAT IHADCB.DCBLRECL
This form is either "modulename.csectname" or "dsectname.fieldname". The absence of a second dot creates this ambiguity. When presented with this form, Z/XDC first checks to see if a load module is present whose name matches the first part of the given name. If so, it then checks to see if a module map of that load module is available. If so, it then checks to see if the load module contains a csect whose name matches the second part of the given name. If any of these checks fail, then Z/XDC checks active dsect maps for the given name combination.

FORMAT SYSCMD.
FORMAT IHADCB.
This form is either "modulename." or "dsectname.". The presence of the single trailing dot prevents Z/XDC from considering equate names or breakpoint names. Also, names of inner objects (such as csect names, label names, and field names) must have at least a leading dot, so those are not considered here either.

FORMAT IMHERE

FORMAT SYSCMD
FORMAT IHADCB
FORMAT AT00001B
This form is either "equatename", "modulename", "dsectname", or "breakpointname". The absence of any dot permits all these possibilities and prohibits from consideration any inner name (i.e. csect names, label names, and field names). Z/XDC searches the permitted possibilities in the following order:
equate name
load module name
active dsect name
breakpoint name

In general, when **leading** name name elements are omitted, then default names are used. Such names are established by the SET QUALIFIER command. See HELP COMMANDS SET QUALIFIER for more information about this.

When trailing name elements are omitted, then the given name is taken to be a reference to the origin of a larger object.

When a period is given all by itself without any name elements, then this is taken to be a reference to the default entry name and csect name as established by the SET QUALIFIER command.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

SEARCHES - Name resolution search order.

EXAMPLES - More examples of symbolic names.

Still more examples can be found at the following topics:

- HELP ADDRESSING SYNTAX BASETERM EXAMPLES

- HELP ADDRESSING SYNTAX OFFSETTERMS EXAMPLES

- HELP ADDRESSING SYNTAX BETWEENTERMS EXAMPLES

- HELP ADDRESSING EXAMPLES

Help ADdressing Parsers Asm RESolution Searches

When Z/XDC is presented with a partially qualified name, it must go to some effort to figure out just what kind of name it is examining. To do this, Z/XDC considers both the form of the given name, the context within which the name is presented, and a name-type search order.

In general, Z/XDC performs searches in the order given below when it is trying to identify and classify a given name. If a name's form does not fit a given classification, then the search in that class is bypassed. For example, "name.name" cannot be an equate name. Also ".name" cannot be a label within a csect if a default entry point name and csect name are not currently defined.

The search order is:
- Equate names.
- The default load module entry point name (established by the SET QUALIFIER command) and symbols within it.
- Any load module entry point name. The search order for entry point names is:
  - The Target Address Space's Job Pack Area (JPA).
  - The System's Modified Link Pack Area (MLPA).
  - The System's Fixed Link Pack Area (FLPA).
  - The System's Pageable Link Pack Area (PLPA).
  - The currently loaded System Nucleus (IEANUC0x).
- Any active dsect name. The dsects are maintained in a LIFO queue. Each time a dsect is created (by DMAP) or assigned a base address (by USING), it is placed at the top of the queue. Consequently, incompletely qualified references to dsect field names are always resolved to the most recently created or based (as appropriate) dsect containing that label.
- Hook names.
- Breakpoint names.
- Absolute storage addresses. If a given name is syntactically identical to a hexadecimal address (example: "FORMAT D345B0"), then it is treated as an address only if it cannot first be resolved as a symbolic name.

Help ADdressing Parsers Asm RESolution Examples

MYMODULE.MYCSECT.MYLABEL
This refers to the location of MYLABEL which is within MYCSECT which is within a load module containing an entry point named MYMODULE.

MYMODULE.MYCSECT.MYLABEL(4)

If MYLABEL is defined multiple times within the MYCSECT csect, then this refers to the fourth occurrence of MYLABEL. (Trust me, it **can** happen.)

MYMODULE.MYCSECT.
MYMODULE.MYCSECT
These refer to the starting location of MYCSECT which is within MYMODULE.

MYMODULE.X#1
This refers to the **physical start** of the load module named MYMODULE.

MYMODULE.
MYMODULE
These refer to the **entry point** of the load module named MYMODULE.

It is important to remember that a load module's entry point and its load point are not necessarily the same place! See HELP ADDRESSING PARSERS ASM EXTENTS for more information.

Suppose that a SET QUALIFIER command has been issued that established MODXYZ as the default entry name and TESTER as the default csect name.

. (all by itself) means "MODXYZ.TESTER". This is the starting address of the TESTER csect within the MODXYZ load module.

.+3BE means +X'3BE' past the TESTER csect's zero point.

.TRTABLE means "MODXYZ.TESTER.TRTABLE". This is the location of the instruction or field labeled TRTABLE within the TESTER csect.

.REPASS. means "MODXYZ.REPASS". This is the starting address of the REPASS csect within the MODXYZ load module.

.PUTPRT.SYSPRDCB means "MODXYZ.PUTPRT.SYSPRDCB". This is the location of the instruction or field labeled SYSPRDCB within the PUTPRT csect within the MODXYZ load module.

More examples of address expressions can be found at the following topics:

- HELP ADDRESSING SYNTAX BASETERM EXAMPLES
- HELP ADDRESSING SYNTAX OFFSETTERMS EXAMPLES
- HELP ADDRESSING SYNTAX BETWEENTERMS EXAMPLES
- HELP ADDRESSING EXAMPLES

Help ADdressing Parsers Asm DUplicates

It is possible for dsect maps and csect maps to contain duplicate names. Accordingly, when a name that has duplicates is used within an address expression, Z/XDC requires that the name be followed by a parenthesized numeric subscript to distinguish this instance of the name from others. (See HELP MAPS for more information about csect and dsect maps.) Examples:

```
FORMAT .OOPS(1)
FORMAT .OOPS(2)
```

Note, subscript (1) normally is used to refer to the first instance of duplicated labels. However, it can also be used to refer to any non-duplicated label. Example: The following commands are equivalent:

```
FORMAT 21C?
FORMAT PSA.PSATOLD?
FORMAT PSA.PSATOLD(1)?
```

When a given name occurs one time in each of several maps, the instances are NOT considered to be duplicates. Each instance can be uniquely referenced by suitable qualification. Examples:

```
FORMAT MYPROG.PUTMSG.SYSPRINT
FORMAT MYPROG.PUTERROR.SYSPRINT
```

The most straight forward cause of duplicate names is programmer error. The Assembler programmer might simply define 2 instances of the same name within a csect or dsect. Doing so, of course, causes the Assembler to report the error; however, it is still possible for the programmer to linkedit and run the resulting code (probably not a good idea but nonetheless possible).

Z/XDC does not yet support using subscripts to distinguish multiple instances of load modules. However, see HELP DEBUGGING TIPS DUPLICATES for comprehensive workaround suggestions.

Help ADdressing Parsers Asm EXtents

In z/OS, load modules and program objects usually reside in storage within single contiguous areas called "extents". IBM's Loader program and some compilers, however, have the ability to create in-storage load modules that reside in two or more noncontiguous extents. Also, the Binder can create programs objects that, when loaded into storage, can reside in multiple extents. A program object having the RMODE(SPLIT) attribute is an example of this. Consequently, Z/XDC recognizes the following syntax for referring to the start of a particular extent of a particular load module.

```
modulename.X#number

Example:  IGC0001I.X#1
```

Z/XDC uses extent numbers in its displays whenever the module being displayed resides in two or more extents.

Z/XDC also displays extent numbers when the main entry point for a module (as indicated by the major CDE) is NOT located at the module's load point.

Help ADdressing Parsers Asm Longnames

"Long names" are symbolic names that are longer that the historical standard of 8 characters. Z/XDC can recognize and process names up to 64 characters long for many kinds of objects. Z/XDC's long name support is available for the following objects:

- Data field names
- Machine instruction names
- Dsect names
- Csect names
- Equates

At this time, Z/XDC does not recognize load modules having names longer than 8 characters.

When writing Assembler Language code, programmers can define field names and statement labels that are as much as 63 characters long. Z/XDC has access to the programmer defined field names via symbol maps that it builds when reading certain output files produced by the Assembler:

- **SYM Records:** These records are produced by all assemblers when PARM='TEST,...' is specified for the assembly.
- **ADATA Records:** These records are produced by all modern assemblers when PARM='ADATA,...' is specified for the assembly.

For more information, see HELP MAPS.

Access by Z/XDC to long names may or may not be possible depending both upon the Assembler used to create the program and upon the output files produced by the Assembler:

- When maps are built from reading ADATA records (produced both by IBM's High Level Assembler, or by Tachyon Software's Cross Assembler, or by Dignus' Systems/ASM), then long names are available for use and display by Z/XDC.
- This is also true when maps are built from **SYM** records produced by Tachyon Software's z/Assembler. (See HELP TACHYON for more information.)

- This is not true, however, for maps built from SYM records produced by any of the IBM assemblers. This is because IBM has adamantly refused to implement the SYM data structure changes needed to support names longer than 8 characters.

To make your program's long names available for display and use by Z/XDC, do either of the following:

Via ADATA using IBM's High Level Assembler:
- Specify PARM='ADATA,...' when running the Assembler.
- Add a //SYSADATA DD card to the assembly JCL. It should point either to a sequential file or to a member of a PDS with the member name being the same as the csect being assembled.
- At the start of the Z/XDC debugging session, use the SET MAPLIBS command to notify Z/XDC of the one or more datasets and libraries that it should search when looking for mapping information.

Via ADATA using Tachyon's Cross Assembler:
- Contact Tachyon Software for details. (www.tachyonsoft.com)

Via ADATA using Dignus' Systems/ASM:
- Contact Dignus, LLC for details. (www.dignus.com)

Via SYM records using Tachyon's Cross Assembler:
- Specify the SYMNAME(XDC) and TEST(XDC) assembly options when you perform your assemblies.

For more information about these options, refer to Tachyon provided documentation. Tachyon Software's website is at www.tachyonsoft.com.

Z/XDC has full support for creating equates with names up to 64 characters long. See HELP EQUATES for more information.

Help ADdressing Parsers Asm Asmequs

ADATA, in addition to a lot of other stuff, contains information about all Assembler equates (EQUs) defined within a program. This includes both relocatable and non-relocatable EQUs. (Note, EQU information is generally not available from SYM data.)

Relocatable EQUs

Relocatable EQUs define labels that reference locations within a csect or dsect.

Examples: Suppose you have some assembly statements such as these:

```
NORMAL_LABEL DS  0H
EQULABEL1    EQU *
EQULABEL2    EQU NORMAL_LABEL
EQULABEL3    EQU NORMAL_LABEL+20
EQULABEL4    EQU NORMAL_LABEL+X'14'
EQULABEL5    EQU .TCBRBP
```

In Z/XDC, these labels can be used in address expressions in the same way as any other label. For example, The following Z/XDC commands all display storage starting at the location referenced by NORMAL_LABEL:


```
FORMAT .NORMAL_LABEL
FORMAT  .EQULABEL1
FORMAT  .EQULABEL2
FORMAT  .EQULABEL3-14
FORMAT  .EQULABEL3-20N
```

Note that the Assembler permits a relocatable equate to be defined that is assigned to a csect or dsect that is different from the one in which the definition appears. The definition of EQULABEL5 (above) is an example of that. This is OK. Z/XDC has the logic necessary to sort out to which section such labels actually belong. So the following two FORMAT commands have the same result:

```
FORMAT .EQULABEL5
FORMAT .TCBRBP
```

Non-Relocatable EQUs

Non-Relocatable EQUs define labels that can be used as numbers or strings. They do not reference storage locations.

Nevertheless, Z/XDC permits you to use non-Relocatable EQUs in address expressions to refer to the place within a map at which the EQU is defined. This makes it easy to view the EQU's definition. For example, here is a snippet from IBM's mapping macro (IKJTCB) for Task Control Blocks:

```
TCBCMP   DS    0BL4 -  TASK COMPL ...
TCBCMPF  DS    B -     INDICATOR  ...
TCBCREQ  EQU   X'80' - A DUMP HAS ...
TCBCSTEP EQU   X'40' - A STEP ABE ...
TCBCPP   EQU   X'20' - SOME PROBL ...
*                     SECOND LOA ...
*                     INDICATED  ...
*                     (OS/VS1)
TCBDMPO  EQU   X'20' - DUMP OPTIO ...
*                     MACRO
TCBSTCC  EQU   X'10' - COMPLETION ...
*                     (OFFSET 17 ...
*                     THIS IS TO ...
*                     COMPLETION ...
TCBNOCC  EQU   X'10' - A COMPLETI ...
```

Once this map has been loaded and assigned to storage, **ANY** of the above displayed labels can be used in address expressions to display the location referenced by the TCBCMP and TCBCMPF labels. Examples:

```
DMAP XDCMAPS.TCB ADATA
USING TCB 21C?
FORMAT .TCBCMP
FORMAT .TCBSTCC
etc.
```

Help ADdressing Parsers Asm Jpalpaplpa

When a load module is read into storage from disk, z/OS keeps track of where that module is and what its attributes are by building a control block and adding it to a queue or a list. The control block that the Operating System builds may be either a "CDE" or an "LPDE" depending upon the following:

LPDE ("Link Pack Directory Entry")
z/OS uses LPDEs to describe load modules located in the PLPA ("Pageable Link Pack Area"). The PLPA and, therefore, the LPDEs are built at system IPL time. LPDEs are contained in a list located in common storage and called the Pageable Link Pack Area Directory. Normally, this directory can be changed only at IPL time.

CDE ("Contents Directory Entry")
z/OS uses CDEs to describe load modules located anywhere in storage other than in the PLPA. This includes the following:

JPA ("Job Pack Area")
This refers to all load modules that are read into a program's private area. The Job Pack Area is described by a JPQ ("Job Pack Queue") which is a queue of CDEs anchored out of the program's jobstep TCB.

LPA ("Link Pack Area")
The PLPA is not the only area of common storage that contains load modules. A second group of common area load modules form what is called the Link Pack Area and is described by an LPQ ("Link Pack Queue") which is a queue of CDEs anchored out of the System's CVT. When a program references a common area load module, the System searches the LPA ∗before∗ searching the PLPA.

MLPA ("Modified Link Pack Area")
FLPA ("Fixed Link Pack Area")
These are actually portions of the LPA, **not** the PLPA! Modules in the MLPA and FLPA usually are read into storage at system IPL time and are described by CDEs on the System's LPQ(**not** by LPDEs in the PLPA Directory). Modules in the MLPA are placed into writable storage, while modules in the FLPA are placed into read-only storage.

If the user uses the name of a load module (or program object) in an address expression, then Z/XDC tries to resolve that name by scanning all available LPDEs and CDEs. Therefore, if the named load module can be found on any Job Pack Queue (JPQ) in the address space, or on the System's Link Pack Queue (LPQ, MLPA, and FLPA), or in the System's Pageable Link Pack Area (PLPA), then Z/XDC will be able to successfully resolve the given name to the module's entry point address in storage.

Note: Usually an address space will have only one JPQ. But it is possible for an address space to have more than one. The Master Scheduler's address space is an example of such an address space.

Note: For a discussion of referring to a module's load point when its entry point address is located somewhere other than at the load point, see HELP ADDRESSING PARSERS ASM EXTENTS.

Note: When CICS brings load modules into storage, it does so using its own internal services, not z/OS services. Therefore, CDEs and LPDEs do not exist for such modules. Accordingly, Z/XDC cannot find such modules. The result is that address expressions that refer to the names of CICS-loaded modules will fail. For information about workarounds for this problem, see HELP DEBUGGING CICS.

Help ADdressing Parsers Asm MOdules

Z/XDC recognizes the names of almost all load modules located in accessible storage. When given a load module name, Z/XDC searches the following areas in the following order.

- The Target Address Space's Job Pack Queue (JPQ).
- The System's Dynamic Link Pack Area (DLPA).
- The System's Modified Link Pack Area (MLPA).
- The System's Fixed Link Pack Area (FLPA).
- The System's Pageable Link Pack Area (PLPA).
- The currently loaded System Nucleus (IEANUC0x).

When multiple instances of a load module reside in storage, normally usage of the name of that module will be resolved (by the above search order) to the first instance encountered. However, when a particular instance of a module has been mapped (see HELP COMMANDS MAP), by-name references to that module will then resolve to the mapped instance, not necessarily the first instance.

When multiple instances of a load module are present, Z/XDC does not have a syntax for explicitly referring to the second or subsequent instance, so if you want to do that, you will have to use other methods. For a comprehensive discussion of this issue, see HELP DEBUGGING TIPS DUPLICATES

Load module names can be used in address expressions in the following contexts.

modulename.
modulename
These two forms refer to the location of a load module's **entry point.** All accessible system queues are searched (as described above) until a CDE or LPDE representing the given module name is found. The first form (with the trailing dot) shown above is less ambiguous than the second.

modulename.csectname.
modulename.csectname
These two forms refer to the location of csect or entry label within a load module. Here, **modulename** may be the primary name or any alias name of a load module. **Csectname** may actually be the name of a csect, **or** the name of a common block, **or** the name of any symbol having the EXTRN or WXTRN attribute.

modulename.csectname.labelname
This form refers to the location of an assembly time address label within a particular csect or common block. **Csectname** map be the name of any csect, common block or **entry label** found within the load module.

In this context, if **csectname** is an **entry label,** then Z/XDC considers that to be a reference to the csect in which that entry label occurs.

modulename.X#number
This form refers to the start of the indicated extent of a load module. Here, **modulename** may be the name or any alias name for a load module.
Note, **entryname.X#1** always refers explicitly to a module's load point without regard to the location of its entry point(s).

For more information, please see HELP MAPS.

Note: When CICS brings load modules into storage, it does so using its own internal services, not z/OS services. Therefore, CDEs and LPDEs do not exist for such modules. Accordingly, Z/XDC cannot find such modules. The result is that address expressions that refer to the names of CICS-loaded modules will fail. For information about workarounds for this problem, see HELP DEBUGGING CICS.

Help ADdressing Parsers Asm DSects

Dsect names can be used in address expressions in the following contexts.

modulename.dsectname
.dsectname
These two forms can only be used with the DMAP command. Modulename indicates which member of a library dataset holds the SYM records or ADATA records that constitute the desired dsect map.

When modulename is omitted (the 2nd form shown above), the load module used by the preceding DMAP command is used by default. If this is the first DMAP command issued during the debugging session, then the default load module, as established by the SET QUALIFIER command, is used. For more information, please see:

```
    HELP COMMANDS DMAP
    HELP COMMANDS SET QUALIFIER
```

dsectname.fieldname
This form refers to the location of a particular field within a particular dsect map. This location depends upon the current value of the dsect's base address.

dsectname.
dsectname
These two forms refer to the current location of the dsect map's origin. A dsect's "origin" is established by the DMAP command and need not coincide with the map's first field name.

.fieldname
This form refers to the current location of a particular field name contained within some unspecified dsect map. Depending upon the command being used, either all dsects (DMAP and USING commands) or all active dsects (DISPLAY, FORMAT, and other commands) are searched in a particular order until a dsect is found which contains the desired field. The dsect search order is created and changed by the DMAP and USING commands. It is displayable by the LIST MAPS command.

Note, when a USING command is issued to assign a base address to a dsect map, the basing address expression can resolve either to the Home Address Space or to any other accessible address or data space. In any case, the targeted address/data space is considered to be a part of the dsect map's base address. Accordingly, whenever the map is referenced in an address expression, both the map's address and its address/data space are returned in resolution of the reference. Example:

```
    SET ASID
    DMAP HASPDOC.HCT
    USING HCT ASID(JES2)HASJES20.X#1
    FORMAT HCT.$JOBQPTR?
```

This sequence of commands causes Z/XDC to display JES2's JQE table. Details:

- The SET ASID command is shown here just to illustrate the fact that JES2's address space is **not** currently targeted by Z/XDC's Foreign Address Space Mode.
- The DMAP command reads a dsect map for JES2's Hasp Control Table from the load module named HASPDOC.
- JES2's HCT is located at the start of the HASJES20 load module located in JES2's address space. This USING commands bases the HCT dsect map at that address in that address space.

Therefore, subsequent address expressions that use the HCT or any of its fields as a base automatically refer to locations in JES2's address space.

Help ADdressing Parsers Asm EQuates

Equate names can be used in address expressions in the following context.

equatename (Note the absence of a period)
This form refers to the location of a symbol created by the EQUATE command. Some equate names are built into Z/XDC or are automatically generated by Z/XDC under various conditions. Some words (such as register names or PSW names) are restricted; they cannot be defined as equate names. For more information, please see:

HELP COMMANDS EQUATE
HELP COMMANDS LIST EQUATES
HELP EQUATES

Note, when an equate name is created, the address expression assigned to the equate can resolve either to the Home Address Space or to any other accessible address or data space. Accordingly, the targeted address/data space is considered to be a part of the equate's address. Therefore, whenever the equate is referenced in an address expression, both the equate's address and its address/data space are returned in resolution of the reference. Example:

```
EQUATE MYSPACE 0~ALET(AR3) D 7FFFFFFF
FORMAT MYSPACE+23B000
```

- The "EQUATE" command defines a label named MYSPACE to reference the logical start of the data space designated by the ALET located in access register AR3. (It does not matter whether or not the data space actually starts at location zero or at location X'1000'.)
- Subsequent address expressions that reference the "MYSPACE" label automatically refer to locations in the data space to which the label has been assigned.

Help ADdressing Parsers Asm REGisters

Z/XDC supports references to the following sets of registers.

```
Error level general registers
Error level access registers
Error level control registers
Retry level general registers
Retry level access registers
Double precision floating point registers
```

See HELP EXECUTIONLEVELS for more information about the error level vs. the retry level.

Error level general registers are named either "ERn" or "nER", where "n" is a decimal number ranging from 0 to 15. They can be referred to collectively as "EREGS".

Error level access registers are named either "EARn" or "nEAR", where "n" is a decimal number ranging from 0 to 15. They can be referred to collectively as "EAREGS".

Error level control registers are named either "ECRn" or "nECR", where "n" is a decimal number ranging from 0 to 15. They can be referred to collectively as "ECREGS".

Retry level general registers are named either "Rn" or "nR", where "n" is a decimal number ranging from 0 to 15. They can be referred to collectively as "REGS".

Retry level access registers are named either "ARn" or "naR", where "n" is a decimal number ranging from 0 to 15. They can be referred to collectively as "AREGS".

Retry level control registers are not made available by the System.

Floating point registers are named "FRn" or nFR, where "n" is either 0, 2, 4 or 6. They can be referred to collectively as "FREGS".

The contents of individual registers can be displayed via the LIST command (e.g., "LIST R3" and "LIST EAR5").

The contents of individual retry level general registers, access registers, and floating point registers can be altered by the ZAP command. Error level registers cannot be altered.

When followed by an indirect operator (% ? or !), general register names and access register names can be used within address expressions to refer to locations within address spaces and data spaces. Examples:

R5?
Refers to the location in the current Target Address Space pointed to by the retry level instance of general register R5.

(The Target Address Space is either the retry level Primary Address Space or the "foreign" address space designated by use of the SET ASID command. See HELP VIRTMEM for more information.)

EAR5?
Refers to the address space or data space designated by the ALET contained within the error level instance of access register AR5. The location within that address/data space is taken from the error level instance of **general** register R5.

Floating point registers cannot be used in address expressions.

The LIST command can be used to display entire register sets. Examples:

LIST REGS - Displays the retry level general registers.
LIST EAREGS - Displays the error level access registers.
LIST FREGS - Displays the floating point registers.

When an address expression uses a register or a PSW as its base, the address space that Z/XDC decides to display may be the Home Address Space, the error level Primary or Secondary Address Space, the retry level Primary or Secondary Address Space, the SET ASID targeted Foreign Address Space, or some other space entirely. There are many factors that Z/XDC has to consider in making this decision. For specific information, see HELP ADDRESSING TARGETSPACES.

Help ADdressing Parsers Asm Psws

Z/XDC supports references to two different views of each of two different PSWs. They are:

- **PSW:** The 64-bit "scrunched" view of the **retry** level PSW
- **PSWE:** The 128-bit "wide" view of the **retry** level PSW
- **EPSW:** The 64-bit "scrunched" view of the **error** level PSW
- **EPSWE:** The 128-bit "wide" view of the **error** level PSW

See HELP EXECUTIONLEVELS for more information about the error level vs. the retry level.

Note, the above listed PSW names are referred to herein as **pswkeywords.**

Basically, **PSW** and **PSWE** are just two different views of the same thing: The **retry level** Program Status Word ("PSW"). Similarly, **PSWE** and **EPSWE** are just two different views of the **error level** PSW. In fact, the two views of each PSW can be used interchangeably within Z/XDC **even when** program execution has wandered above the bar! So, for example, to reference the location to which the retry level PSW points, you can use **PSW!** or **PSWE!,** it doesn't matter. Z/XDC will produce the same result regardless of the range of storage into which the PSW points.

Whenever, Z/XDC receives control due to a breakpoint or a program check (and only when the error level and retry level coincide), Z/XDC automatically adjusts the retry level PSW[E] to point back to the start of the instruction that caused the program check (instead of past it). (This adjustment is not done when Z/XDC receives control due to any ABEND that is not a program check.)

The contents of an [E]PSW[E] can be displayed via the various **LIST pswkeyword** commands. Examples:

```
    LIST PSW
    LIST EPSWE FORMAT
```

The storage pointed to by a PSW can be displayed either via a **FORMAT pswkeyword!** command or via a **WHERE** command. Examples:

```
    FORMAT EPSW!
    FORMAT PSWE!
    WHERE
```

Several fields of the **retry** level PSW can be set via the **SET PSW** command. (The error level PSW cannot be changed.) Example:

```
    SET PSW AMODE=64 CC=CCPLUS STATE=PROBLEM ASCMODE=AR
```

Note, the **SET PSW** and **SET PSWE** commands are aliases of each other. They can be used interchangeably.

The instruction address field of the **retry** level PSW (or PSWE) can be changed by the **ZAP PSW** command (not the SET PSW command). Example:

- **ZAP PSW,addressexpression**

  - It does not matter whether you use **PSW** or **PSWE, it will still work.**

  - It does not matter where in storage the address expression resolves to, it will still work. Example: **ZAP PSW,FEDCBA9876543210** will zap this 64-bit address into the wide PSW even though you used **PSW** when you "should" have used **PSWE.**

However, if the PSW's current **AMODE** conflicts with the address to be zapped in, the zap attempt will be rejected. In other words, if you want to zap in an address not supported by the current AMODE, you first have to use the **SET PSW AMODE=value** command before using the ZAP PSW command.

When a pswkeyword is followed by an indirect addressing operator (%, ?, ! or ~INDIRECT() built-in function), it can be used in an address expression to refer to the location pointed to by that PSW, i.e. the current execution location in the current execution address space.

For most programs, the current execution address space is the Primary Space, and for programs not running in cross memory mode, the Primary Space is the Home Space.

To determine the PSW's execution address space, Z/XDC examines the PSW as follows:

- First, the Dynamic Address Translation (DAT) flag is checked. If DAT is off, then the PSW is executing in real storage (not virtual storage).

- Otherwise (if DAT is on), the PSW's ASC-MODE bits are inspected. If home mode is indicated, then the PSW is executing in the Home Address Space.

- Otherwise, (i.e. for Primary, Secondary, and AR ASC-MODEs), the PSW is executing in the Primary Address Space. (Remember, when the PSW's ASC-MODE bits are set to indicate Secondary or AR Address Space Control Modes, this affects the spaces accessed by machine instruction operands, but in does not change the fact that the instructions themselves are fetched from the Primary Address Space.)

Example: **FORMAT PSW?** displays the next instruction that will be executed by the user's program **regardless** of in which address space that instruction is located. **And** regardless of whether that address is above or below the bar.

When an address expression uses a register or a PSW as its base, the address space that Z/XDC decides to display may be the Home Address Space, the error level Primary or Secondary Address Space, the retry level Primary or Secondary Address Space, the SET ASID targeted Foreign Address Space, or some other space entirely. There are many factors that Z/XDC has to consider in making this decision. For specific information, see HELP ADDRESSING TARGETSPACES.

Help ADdressing Parsers Cee

The C Language Parser parses variable names according to the syntax rules of the C language. Z/XDC may or may not use this parser according to:

- Whether or not it is listed in the currently active Parse Order (displayed/set by the LIST/SET PARSEORDER command).
- If its use is explicitly forced by an address expression tag (CEE:---)

Z/XDC calls language specific parsers when parsing variable names. If Z/XDC calls this parser, then it will either succeed and return the storage location of the resolved variable, or it will fail and return some variation of a Syntax Error message. (The returned storage location can then be adjusted by the balance of the address expression in which the variable reference occurs.)

The C Language Parser is available only when Z/XDC's **c/XDC Feature** is licensed. For more information about c/XDC, see HELP DEBUGGING C.

Help ADdressing Numeric

A numeric storage reference may be given as either a hexadecimal or decimal number having a value that ranges from 0 to (2**31)-1, inclusive.

The syntax of a hexadecimal address is simple. Just type the digits. Special punctuation is not needed. Examples:

```
FORMAT 12354
FORMAT 10
FORMAT C3D022
FORMAT 0C3D022
```

Decimal constants have a syntax that is almost as simple. Just type the digits followed either by the letter N, or by a scaling code: K M G T P or X. (See HELP ADDRESSING NUMERIC SCALINGCODES for a discussion of scaling codes.) Examples:

```
FORMAT 256N  - Displays location X'00000100'.
FORMAT 16N?  - Displays the start of the System's CVT.
FORMAT 4K    - Displays location X'00001000'. (Note that K means 1024 decimal,
               not 1000.)
FORMAT 4096N - Displays the same location.
FORMAT 1000  - Displays the same location.
FORMAT 16M-1 - Displays location X'00FFFFFF'.
```

Numeric addresses always refer to the current Target Address Space:

- In Foreign Address Space Mode, this will be the address space specified by the most recently issued SET ASID command. (See HELP COMMANDS SET ASID for more information.)
- In local address space mode, this will be the retry level instance of the Primary Address Space. (See HELP ADDRESSING TARGETSPACES for more information.)

Disambiguating Ambiguous Names

When a numeric hexadecimal address starts with a digit between A and F, it is syntactically indistinguishable from some symbolic names. Consequently, when Z/XDC encounters such an address, it first attempts to resolve it as a name. If after searching its tables it cannot find a match, it then proceeds to treat the value as a number.

Two real life examples of such names are:

- **E:** The alias name for TSO's **EDIT** command.
- **CDAEED:** The name of an LE module.

So a consequence of not requiring a special syntax for hex numbers is that a command such as **FORMAT CDAEED+4** can produce inconsistent results. For this specific example:

- If the load module named **CDAEED** is present in storage, then the location that is +4 past that module's entry point will be displayed.

- On the other hand, If CDAEED is not currently loaded into storage, then location **X'00CDAEF1'** (equals CDAEED+4) will be displayed. (I will admit that this particular case caused me, personally, no end of confusion one late and tiring Winter's night!)

These two cases not withstanding, load modules having names indistinguishable from hex numbers are not particularly common. So it seems to me that simplifying the syntax of hex numbers is a worthwhile benefit.

In any case, these ambiguous names can be disambiguated via the following methods:

- If you wish to refer unambiguously to the name as an **address,** then precede the name with a leading zero. Example: **0CDAEED**

- If you wish to refer unambiguously to the name as a **module name,** then do one or both of the following:
- Enclose the name in tic marks ('). Example: **'CDAEED'**
- Or trail the name with a dot. Example: **CDAEED.**

Examples:

SET ASID JES2
FORMAT 35D80
These commands first set the JES2 address space to be Z/XDC's Target Address Space, and then display location X'00035D80' of that address space. (The SET ASID JES2 command can be issued only when (a) Z/XDC is debugging an authorized program and (b) System Security permits the current user access to the JES2 address space. See HELP SECURITY for more information.)

SET ASID
FORMAT 35D80
The SET ASID command, with no operand, establishes Local Address Space Mode (i.e. clears Foreign Address Space Mode). The FORMAT command then displays location X'00035D80' in the user program's Home Address Space. (See HELP ADDRESSING TARGETSPACES.)

FORMAT CDAEED+4
FORMAT 0CDAEED+4
FORMAT 'CDAEED'+4
FORMAT CDAEED.+4
These commands will resolve as follows:

FORMAT CDAEED+4
- This command will resolve:

```
      - Either to +4 past the entry point address of a load module named CDAEED,
      - Or to the location X'00CDAEF1',
    Depending whether the CDAEED load module is present in or absent from storage.

    FORMAT 0CDAEED+4
  - This command will always resolve to the location X'00CDAEF1', regardless of
    whether or not the CDAEED load module is present in storage.

    FORMAT 'CDAEED'+4
    FORMAT CDAEED.+4
  - These commands either will resolve to the entry point address +4 of the CDAEED
    load module, or will fail, depending whether the CDAEED load module is present
    in or absent from storage.
```

Help ADdressing Numeric Scalingcodes

In most places where Z/XDC accepts a pure hex value it will accept a scaled decimal value as well. The scaling is indicated by various single letter codes that trail the decimal digits. The codes are:

```
N - The number is not scaled.
K - The number is multiplied by 2**10 (kibi-something)
M - The number is multiplied by 2**20 (mebi-something)
G - The number is multiplied by 2**30 (gibi-something)
T - The number is multiplied by 2**40 (tebi-something)
P - The number is multiplied by 2**50 (pebi-something)
X - The number is multiplied by 2**60 (exbi-something)
```

Examples: The following **FORMAT** commands all display location X'00FFFFFF' (which is in the middle of the System Nucleus, and is where the fabled **The Line** is located. (You know... that famous 24-bit line that everything's either above or below?) Anyway, the following commands illustrate various ways by which that location can be displayed.

```
FORMAT FFFFFF
FORMAT 0000000000FFFFFF

FORMAT 16777215N
FORMAT 16777216N-1

FORMAT 16384K-1
FORMAT 16N*16384K-1
FORMAT 10*16384K-1

FORMAT 16M-1
```

Some Things to Notice

The scaling codes can be given either upcased or locased.

The scaling codes represent every tenth power of **2** (not every third power of 10).

It is a mathematical fact that every tenth power of 2 is "close" to every third

power of 10. So...
- One kibibyte (2**10=1,024) is close to one kilobyte (10**3=1,000).
- One mebibyte (2**20=1,048,576) is close to one megabyte (10**6=1,000,000).
- And so on.

When a scaling code is **not** given, the string is interpreted as hex, not decimal.

Special Case

Z/XDC contains special case logic to permit **16X** (16 pebibytes) in certain cases even though 16X cannot be represented within 64 bits. When 16X is used in an expression, and it is directly followed by a minus sign, the **16X-** is allowed, and zeros are used in the computation in place of the 16X. Then **16X-1** becomes **0-1** which equals FFFFFFFF_FFFFFFFF which equals 16X-1.)

Kibi-what????

Oh, and about those disgusting terms, kibi, mebi, flibi, plibi and blahblahblah-bi... Well, they're just the PC world's way of disambiguating the meanings of kilo, mega, gaga, and so forth. If you care to look deeper, it's all there at **en.Wikipedia.org/wiki/Kibibyte**

So disregarding my personal disdain for them, kibi- and friends are useful terms, so I've decided to suck it up and start using them going forward.

Help ADdressing Dotplus

Creating the "Dot" and "Dot Plus" Address Expressions

The **SET QUALIFIER** command establishes a default load module name and csect name so that references to labels, C/C++ statement numbers (c/XDC only) and offsets **within** a csect can be done more easily. For example, without a default qualifier, to refer to a field name or C/C++ statement number, you would have to preface the name or number with both the load module name and the csect name within which the label or number occurred. Example:

- FORMAT myload.mycsect.mylabel
- FORMAT myload.mycsect+3DB
- FORMAT myload.myceepgm.#786

However, with a default qualifier in place, all you need to do is give the label or C/C++ statement number or offset with a leading dot. Examples:
- FORMAT **.mylabel**
- FORMAT **.+3DB**

- FORMAT **.#786**

The **SET QUALIFIER** command establishes a default load module name and control section name. The command's first operand is an address expression that Z/XDC resolves to a storage location. Z/XDC then determines the name of both the load module (or program object) and control section within which that address falls. Those two names (load

module name and csect name) then are saved for use as defaults in subsequent address expression that start with dots. Examples:

```
S Q PSW?
S Q R15?
S Q name1.name2
F .name3
```

For more information, see **HELP COMMANDS SET QUALIFIER.**

Using the Dot in Address Expressions

When subsequent address expressions start with a **dot,** the default load module name **(lmname)** and perhaps the default csect name **(cname)** are substituted into the address expression ahead of the dot. Examples:

- **FORMAT .TOPLOOP** resolves to **FORMAT lmname.cname.TOPLOOP**
- **FORMAT .OTHRSECT.** resolves to **FORMAT lmname.OTHRSECT.**
- **FORMAT .** resolves to **FORMAT lmname.cname+0**

But a **dot-plus-offset** expression will resolve in either or **two** ways, depending upon whether or not the default csect has a nonzero assembly time origin, and that origin information is known to Z/XDC:

- **FORMAT .+3BE** resolves to **FORMAT lmname.cname+3BE** (when origin information is **not** available)
- **FORMAT .+3BE** resolves to **FORMAT lmname.cname-asmorigin+3BE** (when nonzero origin information **is** available)

If a leading dot is followed by a label, then depending upon certain syntax details, either **lmname** or **lmname.cname** will be prefixed to the given label. For those details, see **HELP ADDRESSING SYNTAX BASETERM.**

On the other hand, if a leading dot is **not** followed by a label, then it will either be given by itself or be given followed by an operator (+ - % ? !). In these cases:

- If the dot is given bare, then it refers to the csect's physical starting address.
- However, if the dot is given with an offset (Examples, **FORMAT .+2B6** or **F .-20**), then the offset will be relative to the csect's **assembly time origin** (not its starting location in storage).

nonzero Assembly Time Csect Origins

Normally, when a csect is assembled, the assembly listings show that csect's origin to be zero (00000000). However, When multiple csects are assembled together in a single assembly, only the first control section is shown with a zero origin. All the remaining csects show nonzero origins.

In these cases, a control section's assembly time zero point precedes the start of that section by a distance equal (of course) to section's starting origin. **Example:** If a csect's first byte is labeled by the Assembler as being at offset 00002B48, then that csect's **assembly time origin** is located -X'2B48' bytes before that csect's physical starting point.

But when a control section has been loaded into storage, both the System and Z/XDC

consider that csect's **execution time origin to be its starting address.**

However, under the following conditions, Z/XDC will learn of a csect's assembly time origin and will, therefore, change the execution time origin to match the assembly time origin. Those conditions are:

- **If** a control section was assembled with a nonzero origin,
- **And if** a map of that control section has been loaded by the **MAP** or **DMAP** command,
- **And if** the control section's assembly time origin is available in the mapping data (ADATA or SYM data) from which Z/XDC built the map,
- **Then** the map's execution time origin will be changed to match its assembly time origin. That will be the control section's starting address in storage **minus** its assembly time origin.
- **And** .+offset address expressions will fail when the given offset is less that the csect's assembly time origin.

By doing this, Z/XDC makes it possible for the user to correlate storage displays of the control section's content with displays of that same control section from his assembly listings.

How Z/XDC Obtains Assembly Time Origin Information

A control section's assembly time origin might not always be available to Z/XDC. It is not available, of course, until a **MAP** or **DMAP** command has been issued to load the map. And even once the map is loaded, assembly time location information might still be unavailable:

- If the map is built from ADATA, then there is no problem. Assembly time starting addresses are always available in ADATA.
- If the map is built from SYM data, then Assembly time origins will normally **not** be available. This is because even though the Assembler does convey the necessary information in the object file that it produces, the Binder fails to preserve that information. However, there are two ways in which you can preserve the information for use by Z/XDC:
  - **XDCSYMED** is a utility that is provided as a part of the Z/XDC product. It reads object files produced by the Assembler and massages them in various ways. One of the things it does is preserves Assembly time csect origins in a place where Z/XDC can find them. For more information, see **HELP MAPS SYM EXCESSSYMDATA XDCSYMED.**
  - **Tachyon Software** makes an Assembler named **z/Assembler** that places the Assembly time csect starting origins in a place where Z/XDC can find them. For more information, see **HELP TACHYON.**

For more information, see **HELP MAPS CSECTMAPS ASMOFFSETS.**

Assembly Origin Information Causes Dot-Plus-Offset Expressions to Change!

One consequence of Z/XDC's usage of assembly time origins is that for control sections having nonzero origins the meaning of **.+offset** address expressions can change!

- **If** the **SET QUALIFIER** command has been issued making such a csect the default csect,
- **And if** that csect has not yet been mapped,
- **Then** that csect's execution time zero point will be its starting address.

However:

- **If** a map of this csect is then loaded via the **MAP** command,
- **Then** that csect's execution time zero point will immediately change to match its assembly time origin.

Help ADdressing Implicit

Z/XDC maintains internally two location pointers. They are a **Current Display Pointer (CDP)** and a **Next Display Pointer (NDP).** Usually, the CDP is the address of the area of virtual storage most recently displayed by Z/XDC, and the NDP is the address of the end of that displayed area. These pointers can be used to address storage implicitly.

The Current Display Pointer

The **CDP** can be used as a base term in address expressions. Its use is indicated when an address expression starts with an operator (+ - % ? or !).

Note that one very commonly used address expression that makes use of the CDP is **"+0".** This expression generally refers to the location most recently displayed. Example:

F R15?;M +0;S Q +0

In this command string:

- The location pointed to by general register R15 is formatted and displayed by the **F R15?** command. This command also sets the CDP to the location pointed to by R15.
- Then the **M +0** command loads both a module map and (possibly) a csect map for the code located at the displayed location.
- Finally, the **S Q +0** command sets the default qualifier to be the names of the load module and csect that contain the displayed and mapped location.

In both the MAP command and the SET QUALIFIER command, the **"+0"** address expression refers to the same location displayed by the FORMAT command. This is because the FORMAT command sets the **CDP value.**

The CDP is also used to determine the starting location for searches made by the **FIND** command under certain circumstances. See HELP COMMANDS FIND for more information.

Next Display Pointer

The **NDP** can be used only by the FORMAT and DISPLAY commands. Its use is signaled by the omission from the command of a first operand (e.g. **F** or **F ,3).** The NDP facilitates the quick display of successive storage locations. The successive displays use the same formatting options as the starting display. For more information, see HELP COMMANDS FORMAT.

The **NDP** is also used by the **DOWN** scrolling command when all of the following are true:

- DOWN is issued from the **Primary Window's** command line.
- The Window is already positioned at the bottom of the scroll area (i.e. the Window is not currently scrolled up).
- The display that currently is in view displays storage.

When all of the above are true, the **DOWN** command automatically converts from scrolling downwards through the scroll area to scrolling down through storage. It accomplishes this by making use of the Next Display Pointer:

- The NDP is used for the start of the display,
- And then it is reset to point past the display.
- Thus, the next DOWN command will display next chunk of storage.

The @CDP Built-in Equate

Z/XDC maintains a built-in equate, named **@CDP,** that labels the location that the Current Display Pointer currently points to. For more information, see HELP EQUATES BUILTIN OTHER.

But be careful! Z/XDC maintains **different CDPs** for each window on the display screen. Thus, an @CDP shown in one window will have nothing to do with the location of the CDP associated with a different window. For more information, see HELP FULLSCREEN WINDOWS.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

CDPNDP - How the Current/Next Display Pointers are set.

Help ADdressing Implicit Cdpndp

In general, the **CDP (Current Display Pointer)** and the **NDP (Next Display Pointer)** are set whenever virtual storage is displayed or altered. The CDP makes it easier for subsequent commands (MAP, TRACE, FORMAT, etc.) to reference recently displayed storage. The NDP makes it easier for subsequent DISPLAY and FORMAT commands either to display subsequent storage or to redisplay altered storage.

(For more complete information about the CDP and NDP, see **HELP ADDRESSING IMPLICIT.)**

Commands That Set the CDP and NDP

Specifically, Z/XDC sets the CDP and the NDP as follows for the following commands.

DISPLAY
The CDP is set to point to the start of the display. The NDP points to the end of the display. The NDP allows subsequent storage to be displayed simply by typing a **D** or an **F** with no operands.

FIND
The CDP is set to point to the start of the last location tested by the FIND command. If the searched-for string was found, then the CDP points to the last instance of the string found, and the NDP points to the end of the generated display. If the string was **not** found before reaching the end of the search range, then both the CDP and the NDP point to the end of that range (for use by **reFIND** processing).

FORMAT
The CDP is set to point to the start of the display. The NDP points to the end of the display. The NDP allows subsequent storage to be displayed simply by typing an **F** or a **D** with no operands.

GO
GOT
GOX
Both the CDP and the NDP are set to point to the resume address just prior to execution being return to the program being debugged. This is significant only if the GO/GOT/GOX command then fails for some reason. Then the resume address can be displayed simply by typing an **F** (for FORMAT) command with no operands.

HOOK
HDEFERRED
These command do not change the CDP directly, but they do cause a hook to be set in user code, and when that hook is eventually executed, both the CDP and NDP will be set at that time to point to the retry level PSW’s resume address.

LIST FIXED
Both the CDP and the NDP are set to point to the start of the display. This allows the same location to be redisplayed in another format simply by using the "+0" address expression.

LIST FLOAT
Both the CDP and the NDP are set to point to the start of the display. This allows the same location to be redisplayed in another format simply by using the "+0" address expression.

LIST SUBPOOLS address
Both the CDP and the NDP are set to point to the specified address. (Note, other forms of the LIST SUBPOOLS command do not affect the CDP and NDP.)

VERIFY
When the VERIFY command is used to test virtual storage (as opposed to a register or a PSW), both the CDP and the NDP are set to point to the first byte tested. This

permits a subsequent ZAP command to reference the same address simply by using the "+0" address expression.

WHERE
The CDP is set to point to the retry level PSW's resume address (as hilighted in the WHERE command's display). Note that this is **not** necessarily the start of the display, but this does simplify references by subsequent commands to the PSW's resume address (e.g. "T +4;GOT", which you might want to use when the resume instruction is a BAL and you want to bypass tracing through the subroutine). The NDP is set to point to the end of the display.

ZAP
When the ZAP command is used to alter virtual storage, both the CDP and the NDP are set to point to the zap target address. This allows the zapped storage to be redisplayed simply by typing a "D" (for DISPLAY) or "F" (for FORMAT) command without operands.

ADEFERRED
AT
ATX
TDEFERRED
TRACE
TRAP
These commands do not change the CDP directly, but they do cause breakpoints to be set in user code, and when those breakpoints are eventually executed, the CDP will be set at that time as follows:

- If no automatic commands are given with a breakpointing command, then when the command's breakpoint is reached during user program execution and control is returned to Z/XDC, a WHERE command is automatically executed. Thus, the CDP is set to point to the retry level PSW's resume address (as hilighted in the WHERE command's display), and the NDP is set to point to the end of the display.

- If automatic commands ∗are∗ given with a breakpointing command, then the CDP and NDP are set according to the automatic commands.

- If none of the automatic commands affect the CDP and NDP, then the CDP and NDP are both set to point to the retry level PSW's resume address.

Temporary CDP Setting Across Multiple Address Expressions Within a Single Command

AT
ATX
SHOW
TRAP
Unlike other CDP setting commands, these commands do not permanently change either the CDP or the NDP. However, they **do temporarily change** these pointers for only the duration of the commands' own processing.

- Each of these commands accepts any number of address expressions as operands. As each address expression is processed, Z/XDC checks to see whether or not the expression uses the CDP as its implied base. If it does, then the CDP remains unchanged by that operand. On the other hand, if the address expression does **not** use the CDP (i.e. has an explicit base term), then the expression **does**

change the CDP (temporarily) for possible use by one or more following address expressions.

- Once the command completes, all CDP changes that may have occurred are discarded, and the CDP is restored to the value it had prior to the command.

- These rules have the following consequences:

  - Commands that execute following an AT/ATX/SHOW/TRAP command will not see any change in the CDP temporarily caused by the AT/ATX/SHOW/TRAP command.

  - A sequence of consecutive address expressions (within the AT/ATX/SHOW/TRAP command) that all have implied base terms will all use the same CDP value.

  - An address expression that has an explicit base term can be used to temporarily set the CDP for use by subsequent address expressions (if any) occurring within the same command.

See below for specific examples of this feature.

Intercepted User Program ABENDs

Whenever Z/XDC receives control due to an ABEND, both the CDP and the NDP are set as follows:
- If the retry level is the same as the error level, then the CDP and NDP are set to point to the retry level PSW's resume address.
- If the retry and error levels are different, then the CDP and NDP are set to point to the address pointed to by the error level PSW.

For more information, see **HELP EXECUTIONLEVELS.**

Examples:

SHOW R5? +10 +20 PSW? +4 +8
Six lines of display are produced. They are:
1st line - Storage pointed to by R5.
2nd line - Storage located at R5?+10
3rd line - Storage located at R5?+20
4th line - Storage pointed to by the retry level PSW.
5th line - Storage located at PSW?+4
6th line - Storage located at PSW?+8
An equivalent command would be: **SHOW R5? R5?+10 R5?+20 PSW? PSW?+4 PSW?+8**

T PSW?+4 +4 +8 +C;GOT
Assuming the PSW is pointing to a BAL instruction, and assuming the next four instructions following the BAL are each 4 bytes wide, then this T command makes use of the CDP to set a family of transient breakpoints onto the first four machine instructions following the BAL. Then the GOT command causes user program execution to resume:
- The first address expression ("PSW?+4") sets a breakpoint at PSW?+4 and temporarily sets the CDP to point to PSW?+4.
- The second address expression ("+4") uses the temporary CDP to set the second

breakpoint at PSW?+8. It does not, however, change the CDP.
- The third and fourth address expressions ("+8" and "+C") use the same temporary CDP to set breakpoints at PSW?+C and PSW?+10, respectively.

An equivalent command would be: **T PSW?+4 PSW?+8 PSW?+C PSW?+10;GOT**

If the BAL instruction jumps to a subroutine, and if that subroutine eventually returns to one of the four instructions following the BAL, then execution will be recaptured by Z/XDC when the subroutine completes.

Note, Z/XDC's factory default definition for PF key F11 uses a more complex TRAP command to accomplish a similar result while relying on fewer assumptions than are relied upon by this example. See **HELP BREAKPOINTS TRACING SUBROUTINES** for more information.

Help ADdressing Targetspaces

When an address expression uses a register or a PSW as its base, the address space that Z/XDC decides to display depends on the following factors:

- Whether Z/XDC is currently in Foreign Address Space Mode (FASM) or Local Address Space Mode (LASM). Note, FASM mode cannot be set unless Z/XDC is running authorized. See HELP VIRTMEM XDCACCESS FASM for more information.

- Whether or not the user's program was running in Cross Memory Mode with the Primary Address Space set different from the Home Address Space. (See HELP DEBUGGING PCROUTINES for more information.)

- Whether the retry level and error level Address Spaces (Primary and Secondary) are the same as each other or are different. (See HELP EXECUTIONLEVELS for more information.)

- Whether or not the address expression refers to or implies a specific ALET or a generic ALET (values 0, 1, or 2. See HELP VIRTMEM ALETS for more information.).

- Whether the specified register or PSW is an error level or retry level register/PSW.

- For PSW/EPSW based address expressions, whether the PSW is in home ASC-MODE or some other ASC-MODE.

- For general register based address expressions, whether the indirection operator is "?" or "%" versus "!". The "?" or "%" indirection operators will select the same space; the only difference is the width of the address (31 bit or 24 bit). The "!" indirection operator will use the addressing mode of the PSW to determine the width, and the ASC-MODE to determine the target space.

NOTE: When Z/XDC is in Local Address Space Mode (i.e. not in Foreign Address Space Mode), and the user program is not in any cross memory mode, then all these rules simply devolve down to references to the Home Address Space.

Error Level References:

In general, unless they are directed to particular address spaces, address expressions that are based upon error level registers and PSWs refer to the error level instances of the Primary or Secondary Address Space. Examples:

SHOW ER5%
Displays, in the error level Primary Address Space, the location pointed to by the 24 lo-order bits of general register R5.

DISPLAY EAR0?
Displays the location pointed to by the error level instance of **general** register R0 but in an address space indicated by the ALET found in the error level instance of **access** register AR0. (Note that Z/XDC processes pointers and ALETs found in R0 and AR0 no differently than for any other register.) The ALET is processed as follows:
X'00000000' - Refers to the error level instance of the Primary Address Space.
X'00000001' - Refers to the error level instance of the Secondary Address Space.
X'00000002' - Refers to the Home Address Space. (Error level vs. retry level is not relevant.)
Anything else - Refers to an address space or data space that must be accessible to the current task in the Home Address Space.

See HELP VIRTMEM for more information.

FORMAT EPSW?
This command shows the location pointed to by the error level instance of the PSW (what Z/XDC refers to as the EPSW). If the EPSW is in primary ASC-MODE, secondary ASC-MODE, or AR ASC-MODE, then that location is in the error level instance of the Primary Address Space. If the EPSW is in home ASC-MODE, then that location is in the Home Address Space.

FORMAT -2

This command displays storage starting 2 bytes prior to the location most recently displayed, located in the same address space or data space most recently displayed. This second command references Z/XDC's Current Display Pointer. See HELP ADDRESSING IMPLICIT for more information.

Retry Level References:

When an address expression starts with a retry level instance of a register, the space displayed depends upon whether Z/XDC is in Foreign Address Space Mode (FASM) or Local Address Space Mode (LASM):

- When Z/XDC is in FASM mode, general register based address expressions refer to the address space to which FASM mode is targeted (the Target Address Space). FASM mode is set by use of the "SET ASID name" command. See HELP COMMANDS SET ASID for more information.

- When Z/XDC is in LASM mode (i.e. not in FASM mode), general register based address expressions refer to the retry level instance of the Primary (or Secondary) address space. LASM mode is set (i.e. FASM mode is cleared) by using the SET ASID command without operands.

Address expressions that use the retry level instances of the generic ALET values 0 (primary) and 1 (secondary), refer to the retry level instances of the Primary and Secondary Address Spaces regardless of FASM mode.

Address expressions that are based upon the retry level PSW refer either to the retry level instance of the Primary Address Space or to the Home Address Space, depending upon the Address Space Control Mode (ASM-MODE) indicated by the PSW:

- If the PSW indicates primary ASC-MODE, secondary ASC-MODE, or AR ASC-MODE, then PSW based address expressions refer to the retry level instance of the Primary Address Space.
- If the PSW indicates home ASC-MODE, then PSW based address expressions refer to the Home Address Space.

Whether or not Z/XDC is in FASM mode does not affect or change the address space referenced by PSW based address expressions.

Examples:

For the following examples, assume that the user program is in cross memory mode (PASID<>HASID), and Z/XDC is in FASM mode (Target Address Space set different from the Home Address Space).

FORMAT PSW?
Displays the next instruction to be executed in the retry level instance of the Primary Address Space.

FORMAT R5?
Displays the location pointed to by the retry level instance of general register R5 in FASM's Target Address Space.

FORMAT R5?~ALET(0)
Displays the location pointed to by the retry level instance of general register R5 in the retry level instance of the Primary Address Space.

FORMAT AR5?
Displays the location pointed to by the retry level instance of **general** register R5 in the address space or data space indicated by the ALET found in the retry level instance of **access** register AR5. The ALET is processed as follows:
X'00000000' - Refers to the retry level instance of the Primary Address Space.
X'00000001' - Refers to the retry level instance of the Secondary Address Space.
X'00000002' - Refers to the Home Address Space. (Error level vs. retry level is not relevant.)
Anything else - Refers to an address space or data space that must be accessible to the current task in the Home Address Space.

See HELP VIRTMEM for more information.

Help ADdressing Functions

Z/XDC has several built-in functions that can be used in address expressions. These functions do such things as change the address space or data space to which the expression is targeted, and extract values from storage for use in the expression's computation. Built-in functions generally have operands that may, themselves, be address expressions.

The following is a summary of the built-in functions that are supported for use within address expressions. See **HELP FUNCTIONS and then a name** for detailed descriptions of these functions.

ALET(...) - Selects a Target Address Space or data space from the cluster of spaces that are accessible to the job, TSO session, or system task targeted by the current Foreign/Local Address Space Mode.

In general, ALET(0) is interpreted as the retry level Primary Space, ALET(1) as the retry level Secondary Space, and ALET(2) as the Home Space. See **HELP VIRTMEM ALETS** for more information.

ASID(...) - (Alias: ASN) Temporarily sets, changes, or terminates Foreign Address Space Mode.

ASID(REAL) - Temporarily sets Real Address Space Mode.

F4(...) - Extracts 4 bytes from storage, from a register or from a PSW, propagates the value's sign bit leftwards out to 64 bits, and makes the result available for use in the computation of an offset term. Note, F4 is an alias of the **S4** function.

H2(...) - Extracts 2 bytes from storage, from a register or from a PSW, propagates the value's sign bit leftwards out to 64 bits, and makes the result available for use in the computation of an offset term. Note, H2 is an alias of the **S2** function.

IEQ(...) - Inserts the value of an IPCS equate into the current address expression at this point. Available when using dump/XDC only.

ILC(...) - Returns the offset to the nth machine instruction past the location calculated so far in the current address expression.

ILIT(...) - inserts the value of an IPCS literal equate into the current address expression at this point. Available when using dump/XDC only.

NXINST(...) - Returns the location of the machine instruction that will be executed **next after** the current instruction pointed to by the retry level PSW. This function sets both the address and address space for the address expression being resolved.

NXSEQ(...) - Returns the address of the nth machine instruction following a given address.

REAL(...) - Temporarily sets Real Address Space Mode and returns the real address (as a location) of the given address expression.

RDDD(...) - Extracts and converts an s-con (a base-displacement) into an address,

and substitutes that for the address expression's "location calculated so far". Works like an indirect operator.

- **S1(...)** thru **S8(...)** - Extracts from 1 to 8 bytes from storage, from a register or from a PSW. It then propagates the value's sign bit out to 64 bits, and makes the result available for use in the computation of an offset term.

- **XADDR(...)** - (Alias: XADR) Resolves a given address expression and returns its resolved value as a numeric that can be used in offset computations. (Makes it possible, for example, to align an address expression to a page boundary.)

- **XALET(...)** - Returns an ALET as a numeric that can be used in offset computations.

- **XASID(...)** - (Alias: XASN) Returns an ASID as a numeric that can be used in offset computations.

- **X1(...)** thru **X8(...)** - Extracts from 1 to 8 bytes from storage, from a register or from a PSW. It then pads it with zeros on the left out to 64 bits, and makes the result available for use in the computation of an offset term.

- Restrictions exist on the placement of function calls within address expressions, and these restrictions are different depending upon which function is used. See **HELP FUNCTIONS CATEGORIES** for specific placement information.

- Function names may be abbreviated down to a unique (unambiguous) truncation. Example: The ASID(...) function can be abbreviated to just **AS(...).** For information about the general syntax of built-in functions, see **HELP FUNCTIONS SYNTAX.**

- Various function calls can be placed almost anywhere within an address expression. In some cases, Z/XDC cannot syntactically distinguish a function name from other names that may occur. Accordingly, any function call may be preceded by a **Function Leader Character** (**FLC** for short) to notify Z/XDC that what follows is the name of a function. In unambiguous situations, the FLC can be omitted. Example: **FORMAT .TCBTIO?+X2(.DCBTIOT).**

- The default FLC character is a tilde (**~**). Example: **FORMAT 03BACD~ALET(00010003).** Without the tilde, Z/XDC would have a hard time determining where the hex address left off and the function call began. For more information, see **HELP FUNCTIONS FLC.**

- For detailed information about all built-in functions, read the series of topics starting with **HELP FUNCTIONS.**

Help ADdressing Syntax

Address expressions have three distinct parts:

- The **base term** defines a starting point for the address expression.
- **Offset terms** start with plus or minus signs and compute values to be added to or subtracted from the starting address.
- **Indirect operators** may occur only between terms. They cause the current

calculated address to be replaced by a value loaded from virtual storage, a register or a PSW.
All of these parts may be omitted under various circumstances.

THE BASE TERM
This defines the starting point of an address expression. It may be a storage address (either absolute or symbolic), a register reference, a PSW reference, certain built-in functions, or omitted. A base term establishes an address expression's initial **location.** (A "location" is defined as being a particular storage address within a particular address space or data space. It is important to understand that the notion of "location" includes both a specific address **and** a specific address/data space.)

If the base term is a register or PSW, then the term **must** be followed immediately by an indirect operator (indicating that the register or PSW is being used as a pointer). This is because all address expressions **must** resolve to locations in storage. Registers and PSWs are not locations in storage.

OFFSET TERMS
These compute values to be added to or subtracted from the location calculated so far. Offset terms start with an unnested plus or minus sign. Offset terms are **not** addresses. They are simply computed numeric values that eventually are added to or subtracted from an address for the purpose of calculating a new address.

INDIRECT OPERATORS
During the evaluation of an address expression, an indirect operator (% ? or ! or the ~RDDD() function or the ~INDIRECT() function) causes a new address to be loaded from the location calculated so far (from a register, a PSW, or storage). This new location replaces the address expression's old "location calculated so far".

In principle, indirect operators may not occur within the base term or offset terms. They may occur only between terms.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

BASETERM - Describes base terms.
OFFSETTERMS - Describes offset terms.
BETWEENTERMS - Describes the indirect operator area between terms.

Help ADdressing Syntax BAseterm

The **base term** of an address expression defines the expression's starting point.

To avoid ambiguity, the base term may be quoted. (for example 'ussprogram01.load'.+124 - in this case the period between ussprogram01 and load is part of the base term whereas the other period separates parts of the addressing expression).

The base term may be:

- A storage address (either hexadecimal, decimal, or symbolic) within an address space or data space. Examples:
  - **10** (hex): Location X'00000010'
  - **10N** (decimal): Location X'0000000A'
  - **16m** (scaled decimal): Location X'01000000'
  - **IEFBR14** (symbolic): The location of the IEFBR14 load module
  - **ussprogram01** (symbolic): The location of a program object loaded By Unix System Services (USS) (Note that only the file name is used - see below)
  - **'upg02.load'** (symbolic): The location of a program object loaded By Unix System Services (USS) (This example shows the use of quotes) (Note that only the file name is used - see below)
  - **"p/upg03.load"** (symbolic): The location of a program object loaded By Unix System Services (USS) (This example shows the use of quotes) (Note that the full path name is used - see below)
  - **IGG0191A.IGG0196A** (symbolic): The location of the IGG0196A csect within the IGG0191A load module (This form can be used only after a suitable MAP command has been issued.)

  (When searching for a matching program name, if the specified name has at least one forward slash (or solidus, '/') then the entire path name is matched. Otherwise just the file name is matched)

- A PSW reference. Examples:
  - **PSW!** - The location pointed to by the retry level PSW
  - **EPSW?** - The location pointed to by the error level PSW

- A register reference. Examples:
  - **R14?** - The location pointed by the low half (i.e. the lo-order 31 bits) of general register 14.
  - **RW15!** - The location pointed by the entirety (i.e. all 64 bits) of general register 15.
  - **AR1?** - The location pointed to by the low half of **general** register 1 but in the space (address space or data space) indicated by **access** register 1.
  - **CRn?** - (where **n** can be certain numbers (but not every number) between 1 and 15. If the given CRn points to a table (segment table, trace table, linkage stack, whatever), then CRn resolves to that virtual or real storage address. Example:
    - **FORMAT CRW15! DATA** -Displays location of the current linkage stack's current entry's descriptor fields.

- Some functions.
  - Some functions can be used as a base term. For example:
    - The NXINST(...) function can be used to set the next executable instruction address as the starting point.
    - The LMOD(...) function can be used to set the entry point of a load module (etc.) as the starting point. (Note that a load module (etc.) name can be used, but the LMOD(...) function can reference program objects with names that cannot be used directly as a starting point).

- Omitted. When the base term is omitted, Z/XDC's **Current Display Pointer** (CDP) is

used as the address expression's starting address.

A base term establishes an address expression's initial "location calculated so far". A **location** is defined as being a particular storage address within a particular address space or data space. It is important to understand that the notion of "location" includes both a specific address **and** a specific address/data space.

If the base term is a register or PSW, then the term **must** be followed immediately by an indirect operator to change the address expression's starting point from the register or PSW to the location in storage pointed to by that register or PSW.

An address expression's initial address space, in most situations, is the user program's Home Address Space. But when Z/XDC is in Foreign Address Space Mode (FASM mode), or when the user program is in cross memory mode (XMS mode), an address expression's initial address space can be any of several spaces other than the Home Address Space. See **HELP ADDRESSING TARGETSPACES** for detailed information.

Most commonly, a base term is given as a symbolic reference to an object located in storage: a load module name, a dsect field name, a program statement label, etc. For detailed information and rules concerning symbolic names, see **HELP ADDRESSING PARSERS ASM.**

A base term also can be given as a hexadecimal or decimal numeric address. For information about using numeric addresses, see **HELP ADDRESSING NUMERIC.**

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

FUNCTIONS - Describes the built-in functions that may be used as a base term.

REGISTERSANDPSWS - Describes how registers and PSWs can be used as base terms in address expressions.

EXAMPLES - Examples and discussions illustrating the use of base terms in address expressions.

Help ADdressing Syntax BAseterm Functions

Z/XDC supports several built-in functions that can be used within address expressions, but only some of those functions are permitted to be used as base terms. Specifically, only those built-in functions that return **locations** (an address within an address/data space) are allowed to be used as base terms. These functions are:

IEQ(...)
This function (usable only if dump/XDC is active) allows you to use an IPCS equate as the starting point.

LMOD(...)
This function allows you to use a load module (etc.) entry address as the starting point. The load module (etc.) name may be one that is not syntactically valid as the starting name on an expression.

NXINST(...)
This function returns the location of the machine instruction that will be executed **next after** the current instruction pointed to by the retry level PSW. This function sets both the address and address space for the address expression being resolved.

NXSEQ(...)
This function returns the address of the nth machine instruction sequentially following a given address.

REAL(...)
This function returns the real address (as a location) of a given address expression.

For detailed information about all built-in functions, see HELP FUNCTIONS.

Help ADdressing Syntax BAseterm Registersandpsws

Z/XDC of course supports building address expressions based on locations pointed to by registers and PSWs. Syntactically, just follow these rules:

- Start the address expression with the name of the register or PSW that you want to use. Examples: **PSW EPSW AR5 CRW15 ER1**
- Follow that name with an indirect operator: **% ? ! ~INDIRECT(...)**
- Optionally follow that with additional terms (offset terms, more indirect operators) to adjust the location that you want to reference.

Using PSWs

Z/XDC recognizes references to two PSWs (Program Status Words):

- **PSW** and **PSWE** reference the retry level PSW.
- **EPSW** and **EPSWE** reference the error level PSW.

For more information, see HELP EXECUTIONLEVELS.

PSWE and **EPSWE** refer to the full 16-byte Program Status Word, while **PSW** and **EPSW** refer to the 8-byte wide **scrunched** views of the 16-byte PSWs. But for address expression purposes, the two views (8-byte vs 16-byte) are treated as being identical. They can be used interchangeably. So, for example, the command **FORMAT PSW!** is perfectly legal even though a scrunched 8-byte PSW cannot contain a 64-bit address. Remember, the 8-byte view is just a view. The underlying actual PSW is always the 16-byte PSWE.

When a PSW reference is used in an address expression, it must be first, and it must be followed by an indirect operator. (See below.)

The storage address being referenced is determined by examining the lo-order 24, 31 or 64 bits (according to the indirect operator that follows the PSW's name) of the selected [E]PSWE. The **space** in which that address is located is determined by the PSW's **Address Space Control Mode** (ASM-MODE) as follows:

ASC-MODE:	**Target Address Space**
Primary:	- The error level or retry level primary address space.
Secondary:	- The error level or retry level **primary** address space.
AR:	- The error level or retry level primary address space.
Home:	- The error level or retry level **home** address space.

Note, the targeted space is the space that the **PSW itself** references, i.e. the space from which the hardware fetches the instruction stream. (For **Secondary ASC-MODE,** data references will be to a different space [the secondary aspace] than instruction fetch references [the primary aspace].)

Using Registers

Z/XDC supports using error level and retry level general registers, access registers and **(certain)** control registers as an address expression's base term. Just use the name of the desired register, and follow it with an indirect operator. (See below.)

Using General Registers

General registers can be referenced by six sets of names (For more information, see HELP COMMANDS LIST GENERALREGISTERS.):

- **Rn** and **ERn** are Z/XDC's names for the lo-order 32 bits of the retry level and error level general registers.
- **RHn** and **ERHn** are Z/XDC's names for the hi-order 32 bits of the retry level and error level general registers.
- **RWn** and **ERWn** are Z/XDC's names for the entire 64 bits of the retry level and error level general registers.

All of these names can be used as base terms, but there are some quirks in the details!

When using a general register as a base term, Z/XDC will use the register's lo-order 24, 31 or 64 bits (according to the indirect operator being used) as the address expression's starting address. Note, Z/XDC will honor the width of the indirect operation (even **64-bit** indirects) regardless of whether the register is referenced by its 64-bit name **(RWn)** or its **lo-order** 32-bit name **(Rn).** In other words, with respect to being used as the base of an address expression, the Rn names are exactly equivalent to the RWn names. Example: The construct R15! **is legal** even though normally R15 would refer only to the register's lo-order 32 bits. In this special case, R15! is treated as if RW15! has been given. (This relaxation of the rules is done for your and my convenience.)

However, with respect to a register's hi-order 32-bit name **(RHn):**

- 24-bit and 31-bit indirects extract from only the high halves of the 64-bit general registers.
- 64-bit indirects are **not** permitted.

Z/XDC determines the address **space** to be referenced as follows:

- If the register being used is an error level register (e.g. ER5?), then the initial target space will be determined by the ASC-MODE of the error level PSW (EPSW).

- If the register being used is an retry level register (e.g. R5?), then the initial target space will be determined by the ASC-MODE of the retry level PSW (PSW).

- If the register name and its following indirect operator are followed either by an **~ASID(...)** built-in function or an **~ALET(...)** built-in function, then the targeted space will be the address space or data space indicated by the function. (See HELP FUNCTIONS ALET for more information.)

Notice! When the PSW (or EPSW) is in AR mode, the default space for a general register reference is **NOT** affected by the corresponding access register. If you wish to reference an access register's space, then you can use the access register (instead of the general register) as the base term. (See below.)

Examples:

ZAP RW15=11223344AABBCCDD
FORMAT R15!
FORMAT RW15!
FORMAT 15R!
Each of these FORMAT commands attempts to display location X'11223344AABBCCDD' in the address space indicated by the retry level PSW's Address Space Control Mode.

FORMAT ER15?+10
FORMAT ERW15?+10
Each of these FORMAT commands attempts to display a location in the address space indicated by the error level PSW's Address Space Control Mode. That location will be X'10' bytes past the location pointed to by the error level register 15.

FORMAT R15%~ALET(EAR3)
The location pointed to by the lo-order bits of retry level register 15, in the address space or dataspace indicated by the ALET from error level access register 3, is displayed.

Using Access Registers

When using an access register as a base term, Z/XDC will do the following:

- The **space** that is referenced will be the space (address space or dataspace) indicated by the ALET within the given access register.

- The **address** that is referenced will be taken from the lo-order 24, 31 or 64 bits of the **general** register that is associated with the given access register.

Examples:

FORMAT EAR3%
FORMAT 3EAR%
FORMAT 3ER%~ALET(EAR3)
All three of these FORMAT commands do the same thing. The address contained in error level general register 3 is displayed from the address space or dataspace indicated by the ALET contained in the error level access register 3.

Using Control Registers

Control registers contain specific hardware control information as defined in IBM's **z/Architecture Principles of Operation** (SA22-7832). Certain control registers (CR0 CR3 CR4 CR6 CR8 and CR9) contain flags and controls, while others contain pointers to various tables (segment tables, trace tables, linkage stacks, etc.) located variously in real and virtual storage.

Those control registers that do contain pointers **(CR1 CR2 CR5 CR7 CR10-15)** can be used as base terms for address expressions. Those that do not, cannot.

For those control registers that point into real storage, Z/XDC must be running authorized in order to use them. Those that point into virtual storage (CR15, for example) can be used even when Z/XDC is running non-authorized.

Specifically, the following control registers can be used as base terms to point to the indicated structures:

- **CRW1** - Primary Space Region Table Origin or Segment Table Origin (in real storage)
- **CRW2** - Dispatchable Unit Control Table (DUCT) Origin (in real storage)
- **CRW5** - Primary Address Space Number Second Table Entry (ASTE) Origin (in real storage)
- **CRW7** - Secondary Space Region Table Origin or Segment Table Origin (in real storage)
- **CRW10** - Program Event Recording (PER) Starting Address (in virtual storage)
- **CRW11** - Program Event Recording (PER) Ending Address (in virtual storage)
- **CRW12** - Current trace table entry address (in real storage)
- **CRW13** - Home Space Region Table Origin or Segment Table Origin (in real storage)
- **CRW14** - Address Space Number First Table Origin (AFTO) (in real storage)
- **CRW15** - Current Linkage Stack Entry Address (in virtual storage)

When the table being pointed to is located in 31-bit storage, the 32-bit versions of the control registers (CRn) can be used instead of the 64-bit versions (CRWn).

For more information, see HELP COMMANDS LIST CONTROLREGISTERS.

Indirect Operators

As noted above, whenever a register name or a PSW name is used as the base term of an address expression, than name **must** be followed by an indirect operator. That is because address expressions always point to storage, while registers and PSWs are not in storage. Therefore, an indirect operator is needed to show that the register

or PSW is being used as a pointer.

The following indirect operators can be used, and they have the following meanings:

- **%** References the location pointed to by the lo-order 24 bits either of a register's contents or of the PSW's instruction address field. The hi-order bits of the register or instruction address are ignored.

- **?** References the location pointed to by the lo-order 31 bits either of a register's contents or of the PSW's instruction address field. The hi-order bits of the register or instruction address are ignored.

- **!** References the location pointed to by all 64 bits either of a register's contents or of the PSW's instruction address field. (Note, the case of **Rn!** is treated as if **RWn!** had been given.)

- **~INDIRECT(...)** uses those bits either of a register's contents or of the PSW's instruction address field as indicated by the ~INDIRECT(...) function's operand. See HELP FUNCTIONS INDIRECT for more information.

Help ADdressing Syntax BAseterm Examples

What follows below are several examples and discussions illustrating the use of **base terms** in address expressions. For the purposes of those examples please assume that the following commands have been issued. (For more information about these commands, follow their crosslinks.)

EQUATE MYTCB 21C%
Defines a label named "MYTCB" to represent the address of the current TCB.

DMAP XDCMAPS.PSA
Reads a dsect named "PSA" from disk.

USING PSA 0
Assigns the PSA dsect to map storage starting at location zero.

LOAD APPL1
Brings a load module named "APPL1" into storage.

MAP APPL1.APPLINIT
Reads from disk both the module map for the APPL1 load module and, if available, a map for the csect named "APPLINIT".

SET QUALIFIER APPL1.APPLINIT
Establishes APPL1 as the default load module and APPLINIT as the default csect.

AT .+35E
Sets a persistent breakpoint at offset X'35E' past the default csect's zero point. The breakpoint is automatically assigned a name such as "AT00001A". In this example, the default csect is APPLINIT, and its zero point is probably the same as its starting point. (However if APPLINIT had been assembled with a nonzero starting offset, then its zero point would be its starting point minus the assembly time offset.)

The following examples illustrate and describe the various base terms that Z/XDC accepts. In addition, these examples also show how Z/XDC resolves names as being load module, csect, dsect, equate, field, and other names. The specific rules for this resolution process are described in HELP ADDRESSING PARSERS ASM RESOLUTION.

FORMAT 21C
FORMAT D593B8
FORMAT 0E
FORMAT 0EDC4019E

The base may be given as a raw hexadecimal number specifying the virtual address to be displayed. Note, if there is a chance that the hexadecimal number might be confused with a name (such as **E,** the alias name for TSO's "EDIT" command or **EDC4019E,** a C language support module) then simply add a leading zero to make the number unambiguous.

FORMAT 540N
FORMAT 13996984N
FORMAT 14N
FORMAT 3989045662N

Base terms may also be given as decimal addresses. These four examples all resolve to the same locations as do the preceding four examples.

FORMAT 4096N (same as 4K)
FORMAT 4K
FORMAT 16M
FORMAT 1024M (same as 1G)
FORMAT 2G
FORMAT 256T
FORMAT 347P
FORMAT 15X

Base terms may also be given as **scaled** decimal addresses. The scaling factors are represented by the single trailing characters shown above. They represent powers of **2** (not powers of 10). For more details, see HELP ADDRESSING NUMERIC SCALINGCODES.

FORMAT 10?
FORMAT 16N?
FORMAT 0+16N?

All three of these examples resolve to the location of z/OS's Communications Vector Table (CVT).

FORMAT MYTCB
FORMAT APPL1
FORMAT PSA
FORMAT AT00001A

A pure name may be the name of an equate, a load module, a dsect, or a breakpoint. Ambiguous names are resolved in that order.

FORMAT E
LOAD E
FORMAT E

Prior to loading the module named **E** into storage, FORMAT E resolves to location 0000000E. But **after** the LOAD command, FORMAT E resolves to the location of the module named E. ("E" is the alias name for TSO's EDIT command.)

FORMAT 0E

This resolves to the location X'0000000E' regardless of whether or not the E load module is present in storage.

FORMAT 'E'
FORMAT E.
These resolve to the location of the E load module. If the E load module is not present in storage, then these commands fail.

FORMAT APPL1.APPLINIT
FORMAT PSA.PSATOLD
A two-part name may be either the name of a csect within a load module (APPLINIT within APPL1) or the name of a field within a dsect (PSATOLD within PSA).

FORMAT APPL1.APPLINIT.SAVEREGS
A three part name can only be the name of a statement or field (SAVEREGS) within a csect (APPLINIT) within a load module (APPL1).

FORMAT .SAVEREGS
FORMAT .PSATOLD
A single-part name preceded by a dot can be either a statement label within the default csect (SAVEREGS within APPL1.APPLINIT) or a field within a dsect (PSATOLD within PSA). Z/XDC tries these possibilities in that order: Z/XDC searches for the name first in the default csect (if any) then in **all** active dsects. The dsects are searched in an order that can be controlled by the **USING** command and that is displayed by the **LIST MAPS** command. The default csect is established by the **SET QUALIFIER** command.

FORMAT .FUZWITS(3)
There are several circumstances under which it is possible for a csect map or dsect map to have multiple definitions of one or more labels. When this occurs, Z/XDC allows you (in fact, requires you) to use a subscript to specify to which instance of the label you are referring. In this example, the default csect (APPLINIT) apparently contains at least three or more occurrences of the label FUZWITS. **FUZWITS(3)** refers to the third such occurrence. For more information, see HELP ADDRESSING PARSERS ASM DUPLICATES.

FORMAT APPL1.
FORMAT PSA.
FORMAT APPL1.APPLINIT.
FORMAT .APPLINIT.SAVEREGS
FORMAT .APPLINIT.
Leading and trailing dots can be used in various ways to reduce Z/XDC's effort in parsing a name. For example:

- If a name includes two dots (as in the last three of the above five examples), then Z/XDC realizes that it has been given some portion of a three-part name, and Z/XDC attempts to resolve that name only by examining load modules that have been suitably mapped.

- If only a single-part name has been given followed by a dot, then Z/XDC knows that it should examine only objects that can have at least two-part names. This would be only dsect maps and load modules. Therefore, Z/XDC would not try to resolve such a name as an equate name or a breakpoint name.

FORMAT .
FORMAT .+25E
If a dot is given unassociated with any name, then Z/XDC considers that to be an

implied reference to the zero point of the current default csect as established by the **SET QUALIFIER** command. Accordingly, in this example (remember the assumptions listed at the beginning of this topic?), **.+25E** refers to offset X'25E' within the APPLINIT csect of the load module named APPL1.

FORMAT +0
FORMAT -2
FORMAT %
FORMAT ?
FORMAT !

When an address expression starts with an operator, then the expression's base term has been omitted. When this occurs, Z/XDC uses the **Current Display Pointer** as the expression's base. (The **CDP** is an internal pointer that Z/XDC uses to remember the previous location most recently displayed or verified or zapped. See HELP ADDRESSING IMPLICIT for more information.) The FORMAT command is one of several Z/XDC commands that sets the CDP.

FORMAT 10
FORMAT ?
FORMAT 10?

After the 1st FORMAT command, the **CDP** points to location X'00000010' (the System's pointer field to its CVT). This allows the 2nd FORMAT command to display the CVT itself. (Of course, you could simply use the 3rd command to display the CVT immediately, if you're not interested in seeing the pointer field first.)

FORMAT R15?
FORMAT 15R?
FORMAT ER12%
FORMAT PSW?
FORMAT EPSW?

Registers and PSWs can be used as base terms; however, when they are so used, they **must** be followed by an indirect operator (% ? or !). The indirect operator is needed to redirect the reference from the register or PSW itself to the storage that the register or PSW **points** to. (Use various of the LIST commands to display the contents of registers and PSWs themselves.)

In addition to a storage location, Z/XDC must also determine the address space that the register or PSW references:

- For PSW and EPSW based address expressions, the address space initially referenced by the expression is implied by that PSW's Address Space Control Mode (ASC-MODE). For **Primary, Secondary,** and **AR** ASC-MODEs, "PSW?" and "EPSW?" refer respectively to the retry level Primary Address Space and to the error level Primary Address Space, while for **Home** ASC-MODE, they both (PSW? and EPSW?) refer to the Home Address Space.

- For general register based address expressions, the address space initially referenced by the expression depends on the current setting of the Foreign Address Space Mode (FASM) facility as established by the SET ASID command. For Local Address Space Mode, an error level register based address expression refers to the error level Primary Space, while a retry level register based address expression refers to the retry level Primary Space. For Foreign Address Space Mode, the designated foreign address space is the initial Target Address Space.

For more information about how Z/XDC chooses the address expression's initial Target Address Space, see HELP ADDRESSING TARGETSPACES.

FORMAT AR5%
FORMAT 5AR!
FORMAT AR9?
Access registers also can be used as base terms. When they are, the ALET within the **access** register is used to select the address or data space to be referenced, but the actual address used is taken from the associated **general** register. Thus, the first example above formats and displays the location pointed to by general register R5 within the address/data space identified by access register AR5.

ALET information:

- Generic ALETs: ALET values of 0 and 1 refer to the user program's retry level or error level Primary Space and Secondary Space, respectively. An ALET value of 2 always refers to the Home Space.

- Whether the ALET values of 0 and 1 refer to an error level space or a retry level space depends upon where Z/XDC finds that ALET value. When the ALET value is extracted from an error level access register, the error level Primary and Secondary Spaces are used (for values 0 and 1 respectively). When the ALET value is extracted from a retry level access register or from any other source, the retry level Primary and Secondary Spaces are used for values 0 and 1.

- When an ALET value of 2 is extracted from any source whatsoever, the Home Space is used (regardless of whether or not the ALET source was an error level access register).

For more information about ALETs, see HELP VIRTMEM ALETS.

Note, if you want to use an access register without reference to the associated general register, then use the ALET(...) built-in function. Example: FORMAT 5BE6D0~ALET(AR5).

FORMAT AR5?
FORMAT R5?~ALET(AR5)
These two commands produce nearly the identical result. They both display the location point to by general register R5 in the data space indicated by access register AR5. The only difference is if general register R5 contains a zero, then the second command will fail with message DBC047E ADDRESS ZERO ENCOUNTERED. (The first command will not fail.)

FORMAT REAL(.PSAPCCAV?+20)
FORMAT .PSAPCCAR?~ASID(REAL)+20
The "REAL(...)" function can be used as a base term. The first command above formats a display that starts 32 (decimal) bytes past the start of the System's PCCA in real storage. The second command above just shows another way to achieve the same result.

FORMAT ASID(JES2)HASJES20.X#1
FORMAT ALET(AR4)R3?
The ALET(...) and ASID(...) functions do not return values. Instead, they change the address space or data space towards which the address expression is targeted. Accordingly, these functions may appear ahead of a base term to change the space that the term references. In the first example, the HCT control block, located in JES2's address space at the physical start of the HASJES20 load module, is displayed. In the second example, the storage pointed to by general register R3 and located in the address/data space designated by access register AR4 is displayed.

More examples of address expressions can be found at the following topics:

- HELP ADDRESSING PARSERS ASM RESOLUTION EXAMPLES
- HELP ADDRESSING SYNTAX OFFSETTERMS EXAMPLES
- HELP ADDRESSING SYNTAX BETWEENTERMS EXAMPLES
- HELP ADDRESSING EXAMPLES

Help ADdressing Syntax Offsetterms

Offset terms compute values to be added to or subtracted from the location calculated so far. Offset terms start with an unnested plus or minus sign and proceed to the next unnested + or - sign, to an indirect operator, or to the end of the address expression, whichever comes first.

Offset terms may contain hexadecimal constants, decimal constants, various arithmetic and boolean operators, certain built-in function calls, and parentheses. They may **not** contain indirect operators. Example:

FORMAT R15%+(1D-3)/2+X3(R14?+5!)%%-37

This address expression contains four offset terms. They are:

```
    +(1D-3)/2
    +X3(R14?+5!)
    -37   (a hexadecimal constant)
    and the "+5" within the inner address expression "R14?+5!"
```

Notice that the minus sign in the first of these offset terms is nested within parentheses, so it does not delimit an offset term.

Also notice that "R14?+5!" is an operand of a built-in function, so it is an inner address expression that has its own offset term ("+5"), but taken as a whole, it forms part of the 2nd offset term of the outer address expression.

It is important to keep in mind that offset terms calculate numeric values, **not** locations. It is only after the evaluation of an offset term is completed that it is then added to or subtracted from the location calculated so far (thus resulting a new location calculated so far). This is why offset terms may not contain indirect operators since those return locations. (A "location" is an address within a space. See HELP ADDRESSING SYNTAX BASETERM for a more complete definition.)

Also, offset terms **may not** contain built-in functions that don't return numeric values: ALET(...), ASID(...), ASN(...), NXINST(...), NXSEQ(...), RDDD(...), and REAL(...).

Offset terms may contain the following built-in functions that **do** return numeric values: H2(...), ILC(...), XADDR(...), XADR(...), XALET(...), XASID(...), XASN(...), X1(...), X2(...), X3(...), and X4(...).

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

CONSTANTS - Explains the syntax of hexadecimal and decimal constants appearing with offset terms.
OPERATORS - Describes the several arithmetic and boolean operators that can be used in an offset term.
FUNCTIONS - Describes the built-in functions that may be used within an offset term.
EXAMPLES - Gives some examples of complex address expressions.

Help ADdressing Syntax Offsetterms Constants

Offset terms may include hexadecimal constants, decimal constants, scaled decimal constants, and various arithmetic and boolean operators for manipulating them.

The syntax of a hexadecimal constant is simple. Just type the digits. Special framing characters are not needed. Examples:

```
FORMAT R5?+5BC
FORMAT R5?+36702
FORMAT PSW?-38?+2C
```

Decimal constants have a syntax that is almost as simple. Just type the digits followed by the letter N. The following examples are the decimal equivalents of the examples shown above:

```
FORMAT R5?+1468N
FORMAT R5?+222978N
FORMAT PSW?-56N?+44N
```

Scaled decimal constants have a syntax that is just as simple. Just type the digits followed one of the following scaling codes, and the resulting value will be multiplied by the scaling factor implied by the code:

```
N - The number is not scaled.
K - The number is multiplied by 2**10 (kibi-something)
M - The number is multiplied by 2**20 (mebi-something)
G - The number is multiplied by 2**30 (gibi-something)
T - The number is multiplied by 2**40 (tebi-something)
P - The number is multiplied by 2**50 (pebi-something)
X - The number is multiplied by 2**60 (exbi-something)
```

For more details, see HELP ADDRESSING NUMERIC SCALINGCODES.

Examples: The following **FORMAT** commands all display location X’00FFFFFF’:

```
FORMAT FFFFFF
FORMAT 16777215N
FORMAT 16777216N-1
FORMAT 16N*1024K-1
FORMAT 16M-1
```

Of course, address expressions can contain a mixture of decimal and hexadecimal constants. Example:

```
FORMAT PSW?-56N?+2C
```

For information about arithmetic and boolean operators, type **HELP *NEXT (F11).**

Help ADdressing Syntax Offsetterms Operators

Offset terms may include hexadecimal constants, decimal constants, various arithmetic and boolean operators, certain built-in function calls, and parentheses. The following operators are supported:

^ Prefix NOT: - Returns the one's complement of its right-side operand.
+ Prefix PLUS: - Returns its right-side operand unchanged.
- Prefix MINUS: - Returns the two's complement of its right-side operand.
& AND: - Returns the logical "and" of its two operands.
| OR: - Returns the logical inclusive "or" of its two operands. Note, this operator can be given either as a solid vertical bar ("|") or a broken vertical bar ("?").
 # XOR: - Returns the logical exclusive "or" of its two operands.
*** MULTIPLY:** - Returns the arithmetic product of its two operands, truncated to 64 bits.
/ DIVIDE: - Returns the arithmetic quotient of its left-side operand divided by its right-side operand. The remainder is discarded.
\ REMAINDER: - Returns the arithmetic remainder of its left-side operand divided by its right-side operand. The quotient is discarded.
+ ADD: - Returns the arithmetic sum of its two operands, truncated to 64 bits.
- SUBTRACT: - Returns the arithmetic difference of its left-side operand minus its right-side operand. The result is truncated to 64 bits.

All intermediate and final results of offset calculations are carried as 64-bit values. When a final computed offset value is added to or subtracted from the location calculated so far, the result is truncated to 64 bits.

In the absence of overriding parentheses, these operations are processed according to the following priorities. Operations having the same priority are processed as they are encountered, from left to right.

PRIORITY	OPERATIONS		
1ST	PLUS	MINUS	NOT
2ND	AND	OR	XOR
3RD	MULTIPLY	DIVIDE	REMAINDER
4TH	ADD	SUBTRACT	

Help ADdressing Syntax Offsetterms Functions

Z/XDC supports several built-in functions that can be used within address expressions, but only some of those functions are permitted within offset terms. Specifically, only those built-in functions that return numeric values are allowed within offset terms. These functions are:

F4(...)
This function extracts 4 bytes from storage, from a register or from a PSW. The value is then padded out to 64 bits by propagating the value's hi-order bit (the "sign" bit) leftwards. The result is then returned for use in the computation of the offset term within which the function call was found. (F4 is an alias of

the **S4** function.)

H2(...)
This function extracts 2 bytes from storage, from a register or from a PSW. The value is then padded out to 64 bits by propagating the value's hi-order bit (the "sign" bit) leftwards. The result is then returned for use in the computation of the offset term within which the function call was found. (H2 is an alias of the **S2** function.)

ILC(...)
This returns the offset to the nth machine instruction past the location calculated so far in the current address expression.

S1(...) thru **S8(...)**
These functions extract from 1 to 8 bytes from storage, from a register or from a PSW. The extracted value's sign bit is then propagated out to 64 bits, and the result is then returned for use in the computation of the offset term within which the function call was found.

XADDR(...) (alias: XADR)
This function accepts an address expression, resolves it to a virtual address, and then returns that address as a numeric value that can be used in an offset term computation.

XALET(...)
This function returns an ALET as a simple numeric value. The operand may be either a register name or an address expression. If a register name is given, then its contents is presumed to be an ALET, and is returned as this function's result. If an address expression is given, then that expression is resolved to a location in an address space or data space. If the resolved location was reached by use of an ALET, then this function returns that ALET as a simple numeric value. Otherwise, this function returns a zero value.

XASID(...) (alias: XASN)
This function returns an ASID as a simple numeric value. The function's operand may be either a keyword, an address space name, or an address expression:
- If it is a keyword, then the ASID implied by that keyword is returned. This could be the ASID of either the Home Address Space or either the retry level or error level instances of the Primary or Secondary Address Spaces.
- If the function's operand is an address space name, then that space's ASID is returned.
- If the function's operand is an address expression, then it is resolved to a location within an address space or a data space. If it resolves to an address space, then the ASID of that address space is returned. If it resolves to a data space, then the ASID is returned of the address space from which the ALET was resolved.

X1(...) thru **X8(...)**
These functions extract from 1 to 8 bytes from storage, from a register or from a PSW. The extracted value is then padded with zeros on the left out to 64 bits. The result is then returned for use in the computation of the offset term within which the function call was found.

For detailed information about particular functions, see **HELP FUNCTIONS name.**

Note that all of the above listed functions accept address expressions for their operands. These inner address expressions may contain **any** function call since they

(the expressions) are considered by Z/XDC to be within the function's environment, not the offset term's environment. Example:

FORMAT R5?+X2(ASID(JES2)HASJES20+38)*8
Here, "ASID(JES2)" is part of the inner address expression that is the operand of the X2(...) function.

If a nonpermitted function call is found during the resolution of an offset term, then either Z/XDC reports a syntax error or Z/XDC considers the offset term to be ended, depending upon whether or not the function call occurs within nesting parentheses. Examples:

FORMAT R5?+38~RDDD(ER)?
This causes Z/XDC to find an s-con located at X'38' bytes past the location pointed to by R5, then to resolve that s-con using the error level general registers, and then to display the resulting location. In this case the RDDD(...) function delimits the "+38" offset term, so it is legal.

FORMAT R5?+(38~RDDD(ER)?)
This causes Z/XDC to report a syntax error. The parentheses cause the RDDD(...) function to be within the offset term, and that is not permitted since the RDDD(...) function returns an address but offset terms can manipulate only numeric values.

For detailed information about all built-in functions, see **HELP FUNCTIONS.**

Help ADdressing Syntax Offsetterms Examples

The following are some examples of complex address expressions.

EQUATE RB#1Z RB#1+~X2(RB#1+8)*8
EQUATE RB#1Z RB#1+X2(RB#1+8)*8
EQUATE RB#1Z RB#1+X2(+8)*8
EQUATE RB#1Z RB#1+8*X2(+8)
All four of these examples accomplish the same result. They define an equate symbol named **RB#1Z** to label the end of the Request Block labeled **RB#1.** Please note the following:

- The **X2(...)** built-in function extracts two bytes from storage for use in the offset computation.
- The **tilde (~)** is the default Function Leader Character. It notifies Z/XDC that a function name follows. It can always be used. It sometimes is required, depending upon context. It is **not** required in any of these examples.
- **RB#1+8** is the 2-byte wide RBSIZE field. This field gives the length of the Request Block in doublewords.
- The ***8** converts the RBSIZE value to bytes.
- In these examples, **RB#1+8** can be shortened to just **+8** which means "plus eight past the location calculated so far" which, in this case, is the location of **RB#1.** This is *regardless* of where within the offset term the **X2(...)** function call occurs.

EQUATE DDTIOT 21C%+C%+X2(MYDCB.DCBTIOT)
Assume that "MYDCB" is a dsect that has been created and assigned to map the fields of a Data Control Block. Assume also that the DCB has been OPEN'd. Then this command defines an equate symbol named "DDTIOT" to represent the Task I/O Table's DD entry associated with the OPEN'd DCB. Please note the following:

- **21C** is the System's PSATOLD field which points to the current TCB.
- Therefore, **21C%** is the address of the current task's TCB.
- **+C** is the offset of the TCBTIO field into the TCB.
- Therefore, **+C%** points to the current task's TIOT.
- For OPEN'd DCBs, the **DCBTIOT** field contains the offset (from the start of the task's TIOT) of the DCB's associated DD entry.
- Therefore, the **+X2(...)** extracts that offset and adds it to the TIOT's address (the location calculated so far).

FORMAT 0+C0|53/C0&53|8*4*(-5D-71N)

```
1st - FORMAT 0+D3   /C0&53|8*4*(-5D-71N)
2nd - FORMAT 0+D3   /40   |8*4*(-5D-71N)
3rd - FORMAT 0+D3   /48     *4*(-5D-71N)
4th - FORMAT 0+2              *4*(-5D-71N)
5th - FORMAT 0+8                *(-5D-71N)
6th - FORMAT 0+8                *(FFFFFFA3-71N)
7th - FORMAT 0+8                *FFFFFF5C
8th - FORMAT 0+FFFFFAE0
9th - FORMAT 7FFFFAE0
```

These examples illustrate the sequence of computations that Z/XDC follows when evaluating an offset term.

```
1st - C0|53 = D3: OR's priority is higher than ADD's.
2nd - C0&53 = 40: AND's priority is higher than DIVIDE's.
3rd - 40|8  = 48: OR's priority is higher than DIVIDE's.
4th - D3/48 = 2:  DIVIDE's priority is the same as MULTIPLY's, so the DIVIDE is done
                  first.
5th - 2*4   = 8:  The MULTIPLY is done next.
6th - -5D = FFFFFFA3: The prefix MINUS's priority is higher than SUBTRACT's.
7th - FFFFFFA3-71N = FFFFFF5C: The parenthesized SUBTRACT operation is done next.
8th - 8*FFFFFF5C = FFFFFAE0: The last MULTIPLY is performed to yield the offset
                  term's final value.
9th - 0+FFFFFAE0 = 7FFFFAE0: Finally, the offset term's value is added to the
                  location calculated so far (0) to yield the final address result
                  which is then truncated to 31 bits.
```

More examples of address expressions can be found at the following topics:

- **HELP ADDRESSING PARSERS ASM RESOLUTION EXAMPLES**
- **HELP ADDRESSING SYNTAX BASETERM EXAMPLES**
- **HELP ADDRESSING SYNTAX BETWEENTERMS EXAMPLES**
- **HELP ADDRESSING EXAMPLES**

Help ADdressing Syntax BEtweenterms

Indirect operators and certain built-in function calls can occur only between terms. They may not be used within a base term or within any offset term.

The indirect operators are %, ?, and !. The functions permitted between terms are ALET(...), ASID(...), and RDDD(...).

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

INDIRECT - Describes the indirect operators.
FUNCTIONS - Discusses the functions permitted between terms.
EXAMPLES - Gives examples.

Help ADdressing Syntax BEtweenterms Indirect

Indirect operators can be used in address expressions to replace the address component of an expression's **location calculated so far.** The address component (and sometimes the space component) is replaced by a new address loaded from storage or from a register or from a PSW.

A **location** is defined as being a particular storage address within a particular address space or data space. It is important to understand that the notion of **location** includes both a specific address **and** a specific address/data space. Example:

FORMAT 10%%+4%
The address expression in this command is **10%%+4%.** It is parsed as follows:

- Before processing the first **%,** the "location calculated so far" is **X'10'** in the current target address space (which in this case happens to be shared to all address spaces). This location is the address of the System's CVT pointer field.

- The first **%** causes Z/XDC to read location **X'10'** and extract from there the CVT's address. Thus after processing the first **%,** the address expression's new "location calculated so far" is the start of the System's CVT.

- The field located at the start of the CVT is **CVTTCBP,** the pointer to the ancient "new/old TCB pointers" located in the PSA (in the fields named PSATNEW and PSATOLD, locations X'218' and X'21C', respectively). Prior to processing the second **%,** the address expression's "location calculated so far" is that of **CVTTCBP.** The second **%** operator changes the "location calculated so far" to be the address of **PSATNEW** (which is the location pointed to by CVTTCBP).

- After the offset **+4** is computed and applied, the "location calculated so far" is advanced by 4 bytes to become the location of **PSATOLD.**

- Finally, when Z/XDC processes the last **%,** the "location calculated so far" is

replaced by the address of the current program's **TCB.** This then becomes the final result of the entire address expression and is what is returned to the FORMAT command handler for display.

Five Indirect Operators, Six Operations

Z/XDC supports six indirect operations. They are:

%	- 24-bit indirect
?	- 31-bit indirect
!	- 64-bit indirect
!	- AMODE-sensitive indirect **(Deprecated!** use INDIRECT(AMODE) instead.)
RDDD(...)	- S-CON indirect
INDIRECT(...)	- general indirect

All of these operations replace the location calculated so far with a new location extracted from the address pointed to by the old location.

The first three of these operations **(% ?** and **!)** share the following common characteristics:

- They all extract the new address from the lo-order portion (24 bits, 31 bits or 64 bits) of the fullword **(%** or **?)** or doubleword **(!)** located at the old location calculated so far.
- That fullword or doubleword need not be aligned.
- The fullword or doubleword can be any accessible location in any accessible address space or data space.
- The fullword or doubleword may be the instruction address field of a PSW.
- The fullword or doubleword may be any error level or retry level general register or access register, including register 0.
- In the case of access registers, the new address actually is extracted from the associated general register, while the access register is used to set (or reset) the address expression's Target Address Space or data space.

% (24-bit indirect)
This extracts the new address from the lo-order 3 bytes of the fullword located at the old **location calculated so far.** The fullword's hi-order 8 bits are ignored.

? (31-bit indirect)
This extracts the new address from the lo-order 31 bits of the fullword located at the old **location calculated so far.** The fullword's hi-order bit is ignored.

SET BANG 64BIT
! (64-bit indirect)
When SET BANG **64BIT** (the factory default) is in effect, then ! acts as a 64-bit indirect operator. It extracts the new address from all 64 bits of the doubleword located at the old **location calculated so far.**

When a **!** (64-bit indirect) is used against a **lo-order** 32-bit register **(Rn** but not RHn), then it is processed as if **RWn!** has been given.

! is not supported with RHn registers.

SET BANG AMODE
! (current AMODE and ASC-MODE indirect)
This usage is deprecated! Use INDIRECT(AMODE) instead.

When SET BANG **AMODE** is in effect, then **!** acts as an AMODE/ASC-MODE sensitive indirect operator. It extracts the new address from the lo-order 24 or 31 bits of the fullword located at the old **location calculated so far.**

The width of the extraction is determined by the addressing mode of either the retry level PSW **(PSW)** or the error level PSW **(EPSW):**

- If the address being extracted is from an error level general register, an error level access register, or the error level PSW, then the AMODE flags of the EPSW controls the extraction width.

- Otherwise, the extraction width is controlled by the retry level PSW's AMODE flags.

If the address being extracted is from either an error level or retry level **general** register, then the target space is determined by the error level or retry level PSW's **ASC-MODE.** Thus:

- If the corresponding PSW (error level or retry level) has **DAT off,** then the target space is REAL storage.

- Otherwise, the PSW ASC-MODE bits are inspected:
  - When the ASC-MODE indicates Primary, the primary space is selected.
  - When the ASC-MODE indicates Secondary, the secondary space is selected.
  - When the ASC-MODE indicates Access Register, the corresponding Access Register determines the target space ALET.
  - When the ASC-MODE indicates HOME, the HOME space is selected.

INDIRECT(...) (all kinds of indirects)
This is a built-in function that performs an indirect operation as directed by its operands. So it can perform any kind of indirect: 24-bit, 31-bit, 64-bit, AMODE-sensitive, whatever. For detailed information, see HELP FUNCTIONS INDIRECT.

RDDD(...)% (s-con indirect)
RDDD(...)?
RDDD(...)!
This extracts the s-con (a machine instruction's base/displacement field) located at the old "location calculated so far", resolves it using a specified set of registers (retry level or error level registers), and makes the result the new "location calculated so far". This function **must** be followed by an indirect operator (% ? or !) to indicate whether the s-con's resolved address is to be truncated to 24 bits, 31 bits or 64-bits.

Final Comments

Note that when the extraction target address bits 1 through 7 are zero, the % and ? indirect operators all process the same logical address. However, when these seven bits are nonzero, the indirect operators behave differently.

When an indirect operator extracts a new address from a location in storage, the address space or data space being referenced does not change. In other words, the storage locations being referenced both before and after the indirect operation are both in the same address space or data space.

On the other hand, when the indirect operator is extracting a new address from

either a register or a PSW (example: FORMAT PSW?), then the new address can be located in any of several possible address spaces:

- The error level Primary Address Space
- The error level Secondary Address Space
- The retry level Primary Address Space
- The retry level Secondary Address Space
- The Home Address Space
- Foreign Address Space Mode's Target Address Space
- An arbitrary address space or data space as designated by an ALET from an access register.

The decision process that Z/XDC goes through to decide the address space takes into consideration many factors. For details, see HELP ADDRESSING TARGETSPACES.

Help ADdressing Syntax BEtweenterms Functions

Z/XDC supports several built-in functions that can be used within address expressions, but only some of those functions can be used between terms. Specifically, only the following functions are allowed between terms:

ASID(...) and **ALET(...)**
These functions change the target address space or data space of the current address expression. In other words, they change the space component of the address expression's "location calculated so far", but they to not change the address component.

These functions can be used both between terms and in front of the base term. Since they do not return numeric results, they cannot be used within offset terms. (Use the XASID and XALET functions instead.)

RDDD(...)
This function operates similarly to an indirect operator. It extracts the s-con (a machine instruction's base/displacement field) from the old "location calculated so far", resolves it using a specified set of registers (retry level or error level general registers), and makes the result the new "location calculated so far". This function **must** be followed by an indirect operator (% ? or !) to indicate whether the s-con's resolved address is to be truncated to 24 bits, or to 31 bits, or according to the PSW addressing mode. Note: Specifying **both** a general register set and the "!" indirection operator may cause the address expression's target space to change as well, according to the applicable PSW's ASC-MODE setting.

For detailed information about all built-in functions, see HELP FUNCTIONS.

Help ADdressing Syntax BEtweenterms Examples

The following are some examples of address expressions that use indirect operators and the nonnumeric built-in functions. NOTE: The default Target Address Space is either the retry level Primary Address Space or Foreign Address Space Mode's Target Address Space (unless an error level register or EPSW is referenced, in which case the default target address space is the error level Primary Address Space). See HELP

ADDRESSING TARGETSPACES for more information.

FORMAT PSW?
This displays the next machine instruction to be executed by the user's program when it resumes execution (pursuant to a TRACE or GO command). Generally (subject to the retry level PSW's ASC-MODE), that machine instruction will be located in the program's Primary Address Space. For normal programs, the Primary Address Space will be the same as the Home Address Space, but for programs running in cross memory mode, that Primary Address Space can be a different address space. For complete information, see HELP ADDRESSING TARGETSPACES.

FORMAT PSW!+2~RDDD(R)%
Assuming that the user program's next instruction is an RS, RX, or SS type instruction (such as L, ST, STM, MVC, etc.), this command resolves that instruction's first (or only) base/displacement field ("s-con") from the retry level Primary Address Space using the retry level general registers. The resolved address is then truncated to 24 bits, and the storage at the resulting location is then displayed.

FORMAT PSW!+2~RDDD(ER)!ALET(AR2)
This is similar to the preceding example. The s-con is extracted from the retry level Primary Address Space, because the retry PSW is referenced (assuming that the retry PSW has DAT ON and is **not** in HOME mode). From there, the error level registers (instead of the retry level) are used to resolve the s-con, and the resulting address may or may not be truncated to 24 bits according to the retry level PSW's AMODE ("!"). This interim result is then redirected by the "!" indirection operator according to the error level PSW's ASC-MODE. However, the built-in function ALET(...) subsequently redirects that address towards the address/data space designated by the retry level access register AR2.

NOTE: An ALET value of 0 would refer to the retry level Primary Space, an ALET value of 1 would refer to the retry level Secondary Space, and an ALET value of 2 would refer to the Home Space. (See HELP VIRTMEM ALETS for more information.)

FORMAT ASID(JES2)HASJES20.X#1
The HCT control block, located in JES2's address space at the physical start of the HASJES20 load module, is displayed.

FORMAT AR3?+20?ALET(AR4)
In this example, assume that general register R3 points to a control block located in a data space designated by access register AR3. Assume further that at offset X'20' of that control block a pointer resides to some data located in another data space designated by access register AR4. This example follows that chain and displays the data located in that other data space. Z/XDC resolves the address expression as follows:

- **"AR3"** is the expression's base term. It causes Z/XDC to set the initial address/data space to be that which is designated by **access** register AR3. It also sets the initial "address" to be the contents of **general** register R3.
- The **"?"** following "AR3" causes Z/XDC to extract the contents of general register R3 thus changing the address expression's "location calculated so far" from the contents of R3 to the storage pointed to by R3 (in the address/data space designated by AR3). NOTE: If the retry level PSW's ASC-MODE is AR-mode **and** the addressing mode is 31-bit, then you could substitute **R3!** for the **AR3?** term.
- **"+20"** advances the "location calculated so far" by 32 (decimal) bytes.
- The **"?"** following "+20" loads, from the location calculated so far, a pointer and makes it the new location calculated so far. It does **not** change the targeted address/data space from that previously established by the "AR3".

- **"ALET(AR4)"** does not change the address component of the location calculated so far, but it does change the space component to that designated by access register AR4. (General register R4 is not used or referenced by this ALET(...) function.)

REMEMBER the following:
- An ALET value of 0 refers to the error level or retry level Primary Space.
- An ALET value of 1 refers to the error level or retry level Secondary Space.
- An ALET value of 2 refers to the Home Space.
- When Z/XDC obtains an ALET from an error level access register ("EARn"), then if the ALET value is 0 or 1, then the Primary and Secondary Spaces being referred to are relative to the error level environment.
- When Z/XDC obtains an ALET from any other source, then ALET values of 0 and 1 refer to the Primary and Secondary Spaces relative to the retry level environment.
- All other ALET values (including the Home Space referencing value of 2) refer to spaces independently of the retry/error level environment.

For more information, see HELP VIRTMEM ALETS.

More examples of address expressions can be found at the following topics:
- HELP ADDRESSING PARSERS ASM RESOLUTION EXAMPLES
- HELP ADDRESSING SYNTAX BASETERM EXAMPLES
- HELP ADDRESSING SYNTAX OFFSETTERMS EXAMPLES
- HELP ADDRESSING EXAMPLES

Help ADdressing Examples

In the examples below, please assume that symbol tables for the System Nucleus, the CVT, and the current TCB have been loaded and appropriately assigned to storage locations. This can be accomplished as follows.

```
    MAP IEANUC01
    DMAP XDCMAPS.CVTFIX
    DMAP XDCMAPS.TCBFIX
    USING .CVTMAP,10?
    USING .TCB,.CVTTCBP?+4?
```

The MAP command loads a module map for the System Nucleus. The DMAP commands load dsect maps for the CVT and TCB. ("CVTFIX" and "TCBFIX" are the names appearing on DSECT statements generated by the IBM provided mapping macros for these two control blocks. See IBM's "MVS Data Areas" manuals, vols. 1-5.) The USING commands assign base locations for the dsects. Note the address expressions used in these commands.

10??+4?
CVTFIX.CVTTCBP?+4?
.CVTTCBP?+4?
21C?
These address expressions all point to the current TCB.

.TCBSTAB!
TCBFIX.TCBSTAB!
If the current addressing mode is 24-bit, then these point to the current STAE Control Block (an SCB). However, if the current addressing mode is 31-bit, then these expressions fail to point to the SCB because of flag bits located in the hi-order byte of TCBSTAB.

.TCBSTAB%

This points to the current STAE Control Block *regardless* of the user program's current addressing mode (AMODE). This is because "%" loads a 24-bit address thereby causing the flag bits in the hi-order byte of TCBSTAB to be ignored.

.TCBSTAB?
This fails to point to the current STAE Control Block due to the flag bits located in the high-order byte of the TCBSTAB field.

IEANUC01.SVCTABLE+2B*8?
This points to the entry address for SVC 43, the CIRB SVC.

IGC0001I
This points to the entry point for the system's OPEN SVC routine.

IGC0001I.X#1
This points to the load point of the load module that contains the entry point for the System's OPEN SVC routine.

EPSW?-2
This points to the second byte preceding the address pointed to by the error level PSW.

DMAP TCBJSTEP,.TCBRBP
USING TCBJSTEP,.TCBJSTCB%
This creates a second TCB map named TCBJSTEP and assigns its base so that it maps the jobstep TCB.

.TCBJPQ%????????????????????????????????
This searches for the end of the CDE queue that describes the Job Pack Queue. An error message is generated that indicates which indirect operator (?) attempted to load a zero value. The last CDE can then be displayed by using fewer question marks.

DISPLAY PSW?
FORMAT +0
FORMAT
The DISPLAY command creates a raw hex-text (dump like) display of storage pointed to by the retry level PSW. The first FORMAT command, by referring to the CDP (the "Current Display Pointer"), redisplays the same storage in a formatted mode. The second FORMAT command makes use of the NDP (the "Next Display Pointer") to display storage directly following the storage shown by the first FORMAT command.

More examples of address expressions can be found at the following topics:
- HELP ADDRESSING PARSERS ASM RESOLUTION EXAMPLES
- HELP ADDRESSING SYNTAX BASETERM EXAMPLES
- HELP ADDRESSING SYNTAX OFFSETTERMS EXAMPLES
- HELP ADDRESSING SYNTAX BETWEENTERMS EXAMPLES

Help FUNctions

"Built-in" functions can be used for a wide variety of purposes in Z/XDC commands. They are used primarily in address expressions (see **HELP ADDRESSING),** but they also can sometimes be used in other contexts:
- Conditional expressions for breakpoints. (see **HELP BREAKPOINTS CONDITIONAL.)**

- Targets of the **VERIFY** command. (see **HELP COMMANDS VERIFY.)**

Most built-in functions are used within address expressions. They can be used for such purposes as:

- Determining a location (address and address space) in some special way such as:
  - The real storage location of a given virtual address.
  - The location of the machine instruction to be executed next after the current instruction (taking into account possible branching and space switching that may occur).
  - The location of the nth machine instruction sequentially following a given address.
  - The location (address and address space) of the PC routine invoked by a particular PC number.
- Changing ASID or ALET associated with an address expression (i.e. changing the space which the expression references).
- Extracting a value from storage for use in a calculation or comparison.

Some functions return a pure numeric result that either can be used arithmetically in an address expression calculation, or can be tested against in various ways such as these:

- In a conditional breakpoint’s conditional expression.
- As the test target of a VERIFY command.

More Information

It is important to clearly understand that different functions return different **kinds** of outputs, and the kind of output returned by a function strongly affects the contexts within which that function can be used.

Subtopics Index

The following detailed information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

General Information

CATEGORIES - Information about which built-in functions fall into what categories

and how that affects the ways in which they can be used.

CHART - A placement chart summarizing the information given in CATEGORIES: Which functions can be used in what contexts.

RECURSION - Most function calls use address expressions as operands which themselves may contain function calls, and those functions may also have address expressions as operands.

SYNTAX - The general syntax of built-in functions.

FLC - The usage and maintenance of Function Leader Characters.

The Functions

ALET - Changes the space being referenced by an address expression. The function's input is an Access List Entry Token (ALET). The new space may be an address space or a data space, depending upon the ALET.

ASID - Changes the space being referenced by an address expression. The function's input is a keyword, a jobname, or an Address Space ID (ASID). The new space may be any address space, or it may be real storage. (It may not be a data space.)

ASN - Alias of ASID.

F4 - Alias of S4.

H2 - Alias of S2.

ILC - Returns the offset to the nth machine instruction past the location calculated so far in the current address expression.

IEQ - inserts the contents of an IPCS equate at this point in the current address expression. Note that this function is only available when using dump/XDC.

ILIT - inserts the contents of an IPCS literal equate at this point in the current address expression. Note that this function is only available when using dump/XDC.

INDIRECT - Works like an indirect operator, only better. Causes the address expression calculated so far to be replaced by a pointer loaded (in various ways) from storage.

LMOD - return the entry point of a load module (or program object).

NXINST - Returns the location of the machine instruction that will be executed **next after** the current instruction pointed to by the retry level PSW. Possible branching and space switching are taken into account.

NXSEQ - Returns the location of the nth machine instruction sequentially following a given address.

PCLOCATE - Examines a given PC number and returns the location (both address space and virtual address) of the PC routine that would be called by that number.

RDDD - Extracts and converts an s-con (a base-displacement) into an address, and substitutes that for the address expression's "location calculated so far". Works like an indirect operator.

REAL - Returns the real storage location of a given virtual location.

SAVE4 - Saves an ALET value during the evaluation of an address expression.

S1 thru **S8** - These functions extract from 1 to 8 bytes from storage, from a register or from a PSW, propagate its hi-order bit (its sign bit) leftwards out to 64 bits, and make the result available for use in the computation of an offset term.

WIDTH - Trims or pads a numeric value to a specified bit width.

XADDR - Resolves a given address expression and returns its resolved address as

a pure numeric value.

XADR - Alias of XADDR.

XALET - Determines or extracts an ALET and returns it as a pure numeric value.

XASID - Determines or extracts an ASID and returns it as a pure numeric value.

XASN - Alias of XASID.

X1 thru **X8** - These functions extract from 1 to 8 bytes from storage, from a register or from a PSW, pad it with zeros on the left out to 64 bits, and make the result available for use in the computation of an offset term.

ZLEFT - zeros a specified number of a value's left-most bits.

ZRIGHT - zeros a specified number of a value's right-most bits.

Help FUNctions CAtegories

The following lists categorize Z/XDC's various built-in functions according to output types, and they explain the differing contexts in which they can be used.

Many of the built-in functions are used in address expressions, but they all have restrictions upon in what parts of an address expression they can appear. Those restrictions are described below. For an understanding of the detailed structure of address expressions, see **HELP ADDRESSING SYNTAX.**

Built-in functions fall into the following categories:

Location Functions

These are functions that return locations, i.e. specific addresses within specific address spaces or data spaces. They can be used only as the base term of an address expression. (See **HELP ADDRESSING SYNTAX BASETERM** for more information.) The following location functions are supported:

LMOD(...)
Returns the entry point of a named load module (or program object). This function sets both the address and address space for the address expression being resolved.

NXINST(...)
Returns the location of the machine instruction that will be executed **next after** the current instruction pointed to by the retry level PSW. Possible branching and space switching are taken into account. This function sets both the address and address space for the address expression being resolved.

NXSEQ(...)
Returns the location of the nth machine instruction sequentially following a given address.

REAL(...)
Returns the real storage location of a given virtual location.

IEQ(...)

Uses an IPCS equate value as the base of an address expression.

Space Setting Functions

These are functions that change the address space or data space that an address expression is referencing, without directly affecting the expression's numeric address calculation. These functions can only be used in address expressions and only within those areas of an expression referred to as "between terms". (See **HELP ADDRESSING SYNTAX BETWEENTERMS** for more information.) The following space setting functions are supported:

ALET(...)
Changes the space being referenced by an address expression. The function's input is an Access List Entry Token (ALET). The new space may be an address space or a data space, depending upon the ALET.

ASID(...) (Alias: **ASN(...)**)
Changes the space being referenced by an address expression. The function's input is a keyword, a jobname, or an Address Space ID (ASID). The new space may be any address space, or it may be real storage. (It may not be a data space.)

Redirect Functions

These are functions that (like indirect operators) redirect an address expression to reference an abruptly different location. They can only be used in address expressions and only within those areas of an expression referred to as "between terms". (See **HELP ADDRESSING SYNTAX BETWEENTERMS** for more information.) The following redirection functions are supported:

RDDD(...)
Extracts and converts an s-con (a base-displacement) into an address, and substitutes that for the address expression's "location calculated so far". Works like an indirect operator.

INDIRECT(...)
Works like an indirect operator (% ? !). Extracts a new address from the storage location pointed to by the address calculated so far. Supports all AMODEs and a variety of other features.

Numeric Functions

These are functions that return a pure numeric result that can be used both in an address expression's numeric calculation process and in value testing contexts such as:

- A breakpoint's conditional expression. (see **HELP BREAKPOINTS CONDITIONAL.)**
- The **VERIFY** command. (see **HELP COMMANDS VERIFY.)**

When used in an address expression, a numeric function can only be used within those areas of an expression referred to as "offset terms". (See **HELP ADDRESSING SYNTAX OFFSETTERMS** for more information.) The following numeric functions are supported:

XADDR(...) (Alias: **XADR(...)**)
Resolves a given address expression and returns its resolved value:
- As a 64-bit unsigned binary number for use in address expression calculations.
- As a 8-byte wide string for use in testing contexts.

XALET(...)
Determines or extracts an ALET and returns its numeric value:
- As a 32-bit unsigned binary number for use in address expression calculations.
- As a 4-byte wide string for use in testing contexts.

XASID(...) (Alias: **XASN(...)**)
Determines or extracts an ASID and returns its numeric value:
- As a 16-bit unsigned binary number (left-padded with zeros out to 64 bits) for use in address expression calculations.
- As a 2-byte wide string for use in testing contexts.

Limited Numeric Functions

These are functions that return a numeric value for use in an address expression's numeric calculation process. They can only be used in address expressions and only within those areas of an expression referred to as "offset terms". (See **HELP ADDRESSING SYNTAX OFFSETTERMS** for more information.) They cannot be used outside of address expressions.

A "limited numeric function" is "numeric" because it returns a pure numeric result, but it is "limited" because it can be used only within address expressions. (Normal numeric functions can be used both within address expressions and in value testing contexts such as breakpoint conditional expressions and the **VERIFY** command. See above.) The following limited numeric functions are supported:

ILC(...)
Returns the offset to the nth machine instruction past the location calculated so far in the current address expression.

Extraction Functions

These are functions that extract a value from storage for use in an address expression's numeric calculation process. They can only be used in address expressions and only within those areas of an expression referred to as "offset terms". (See **HELP ADDRESSING SYNTAX OFFSETTERMS** for more information.) The following extraction functions are supported:

S1(...)
S2(...) (Alias: **H2(...)**)
S3(...)
S4(...) (Alias: **F4(...)**)
S5(...)
S6(...)
S7(...)
S8(...)
Extracts from 1 to 8 bytes of data from storage, from a register or from a PSW.

It then propagates the value's sign bit leftwards out to 64 bits, and makes the result available for use in the computation of an offset term.

X1(...)
X2(...)
X3(...)
X4(...)
X5(...)
X6(...)
X7(...)
X8(...)
Extracts from 1 to 8 bytes of data from storage, from a register or from a PSW. It then pads the value with zeros on the left out to 64 bits, and makes the result available for use in the computation of an offset term.

Address Adjustment Functions

These are functions that adjust the numeric value of the address (calculated so far) by means other than the addition or subtraction of an offset. They can be used both in address expressions and value testing contexts. They may not occur within offset terms, they may occur only between terms. (See **HELP ADDRESSING SYNTAX BETWEENTERMS** for more information.) The following address adjustment functions are supported:

WIDTH(...)
Trims a numeric value (such as an address) to a specified bit width (from 1 to 64 bits wide). The result is then padded on the left with zeros out to 64 bits.

ZLEFT(...)
ZRIGHT(...)
These zero out a specified number of bits of a numeric value (such as an address). the zeroing occurs on the left or the right, respectively. Note, the width of the number is not not changed. It remains at 64 bits. Only its hi-order or lo-order bits are zeroed.

String Value Functions

These are functions that return a string value that is inserted into an address expression.

ILIT(...)
Inserts the value of an IPCS literal equate into an address expression.

String Functions

These are functions that return a string result. They cannot be used in address expressions. They can only be used in value testing contexts such as:

- A breakpoint's conditional expression. (See **HELP BREAKPOINTS CONDITIONAL.)**
- The **VERIFY** command. (See **HELP COMMANDS VERIFY.)**

No string functions currently exist. We expect to write support for such in a future Z/XDC version or release.

Most of the built-in functions not only can appear within address expressions, they also can accept address expressions as operands, and those address expressions can themselves contain built-in functions that also have address expressions as operands. In other words, address expressions and built-in functions can be nested **within** each other recursively to any depth! See **HELP FUNCTIONS RECURSION** for information about the relationship between recursion and the placement rules for function calls.

For a chart summarizing the above information, type **HELP *NEXT** (F11).

Help FUNctions CHart

The following chart summarizes the information given in the preceding topic **(HELP FUNCTIONS CATEGORIES).** For more information, see **HELP *BACK.**

```
                                +-----------+-----------+
                                |   P E R M I T T E D   |
                                |  P L A C E M E N T S  |
                                +-----------+-----------+
                                |  ADDRESS  |  TESTING  |
                                |EXPRESSIONS|  CONTEXTS |
                                +---+---+---+-----+-----+
                                |   | B |   |     |     |
                                |   | E | O |     |     |
                                |   | T | F | C B |     |
                                | B | W | F | O R |     |
                                | A | E | S | N E |     |
                                | S | E | E | D A |     |
                                | E | N | T | I K | V C |
                                |   |   |   | T P | E O |
                                | T | T | T | I O | R M |
                                | E | E | E | O I | I M |
                                | R | R | R | N N | F A |
                                | M | M | M | A T | Y N |
                                |   | S | S | L S |   D |
+------------------------------+---+---+---+-----+-----+
| Location Functions |   |   |   |     |     |
|   LMOD(...)                  | Y | N | N | NO  | NO  |
|   NXINST(...)  REAL(...)     | Y | N | N | NO  | NO  |
|   NXSEQ(...)   PCLOCATE(...) |   |   |   |     |     |
|   IEQ(...)                   | Y | N | N | NO  | NO  |
+------------------------------+---+---+---+-----+-----+
| Space Setting Functions |   |   |   |     |     |
```

ASID(...)	N	Y	N	NO	NO
ALET(...)					
Redirection Functions					
RDDD(...) INDIRECT(...)	N	Y	N	NO	NO
Numeric Functions					
XADDR(...) XASID(...)	N	N	Y	YES	YES
XALET(...)					
Limited Numeric Functions					
ILC(...)	N	N	Y	NO	NO
Extraction Functions					
S1(...) --- S8(...)	N	N	Y	NO	NO
X1(...) --- X8(...)					
Address Adjustment Functions					
ZLEFT(...) ZRIGHT(...)	N	Y	N	YES	YES
WIDTH(...)					
String Value Functions					
ILIT(...)	Y	Y	Y	NO	NO
String Functions					
none	N	N	N	YES	YES

Help FUNctions RECursion

Many of the built-in functions accept address expressions as operands. These operands are referred to as "inner address expressions", and they may be just as simple or complex as any other address expression. No special restrictions exist for inner address expressions.

In fact, inner address expressions may themselves contain function calls whose operands too can be address expressions. There is no specific limit on how deep such recursions can go. The recursion of address expressions does, however, give rise to the following considerations.

The Effects of Recursion on the Placement of Built-in Functions

As described in HELP FUNCTIONS CATEGORIES, many restrictions exist regarding the placement of functions within various parts of address expressions. It is important to understand that such restrictions apply only at an address expression’s current recursion level. They do not create restrictions regarding the placement of the current address expression within an outer address expression. Example:

FORMAT TABLE+XASID(NXINST()+2C)*2
The address expression "TABLE+XASID(NXINST()+2C)*2" contains "NXINST()+2C" as a nested inner address expression. NXINST(...) is restricted to being used only as an address expression’s base term, and within the inner expression, that is

exactly what it is: a base term. This restriction does not, however, prevent the inner address expression as a whole from being used within an offset term of the outer address expression: "TABLE+XASID(...)*2

The Effects of Recursion on the Use of the Current Display Pointer (CDP)

When an outer address expression starts with an operator (e.g. FORMAT +0), the implied base term of the address expression is the Current Display Pointer (CDP). However, When an **inner** address expression uses an implied base term, the "location calculated so far" from the **next outer** address expression is used instead of the CDP. (A "location" is defined as being a particular storage address within a particular address space or data space.) Example:

EQUATE RB#1Z RB#1+X2(+8)*8
The X2(...) function has "+8" for an operand; therefore, "+8" is an inner address expression. At the time that Z/XDC encounters the X2(...) function call, the "location calculated so far" in the outer address expression is the address of the RB#1 label. Accordingly, that address is used by the inner address expression as its implied base. Thus the entire address expression is equivalent to "RB#1+X2(RB#1+8)*8".

During the evaluation of an inner address expression, that expression's own instance of a "location calculated so far" is created. When the resolution of the inner address expression completes, the resulting address is returned to the calling function for use there; however, the resolved value of an inner address expression does not directly affect the "location calculated so far" in the outer address expression.

However, depending upon the specific function, an inner address expression's value can have indirect effects on the outer expression's "location calculated so far". For example, the ASID(...) function can change the outer address expression's Target Address Space, depending upon the resolved value of the function's operand.

Help FUNctions SYntax

Built-in functions may be used within address expressions and within various testing contexts. See HELP FUNCTIONS CATEGORIES for more information.

All built-in functions conform to the following general syntax:

~name(operands)

~ (tilde)
This is the "Function Leader Character" (FLC for short). Its presence informs Z/XDC that a function name follows. It is always permitted, but it may or may not be required depending upon the context in which the function occurs: If the function name immediately abuts following some other name, then the FLC is required to separate the two names. See HELP FUNCTIONS FLC for more information.

name(
The names of all supported built-in functions are alphanumeric with the first character being alphabetic. The name **must always** be followed by an open parenthesis (). This is true even if no operands are given. Example: **FORMAT**

NXSEQ()

(operands)
Built-in functions accept zero, one, or more operands according to the following syntax rules:

- If multiple operands are given, they must be separated from each other by commas or blanks:
  - A separation of n blanks is considered to be a single blank.
  - A separation of n commas is considered to be a n commas.

- Extra leading or embedded commas (not blanks) must be used to indicate the omission of leading or embedded operands. Trailing commas may (but need not) be used for omitted trailing operands.

- The open and close parentheses **must always** be given even when all operands are omitted. Example: **FORMAT NXSEQ()**

Note that a large number of the supported built-in functions accept address expressions as operands, and those address expressions themselves may contain built-in functions that also have address expressions as operands (and so on and so on). Z/XDC fully supports the recursion process necessary to resolve address expressions nested to any level.

Detailed information about specific operands is given in the several topics specifically describing the various functions. See "HELP FUNCTIONS name" for specific information.

Help FUNctions FLc

In some contexts, the name of a function cannot be syntactically isolated from preceding command text. Example: **FORMAT PRIVATEASID(JES2).** Here, Z/XDC cannot syntactically isolate the equate named PRIVATE from the function named ASID. To resolve this problem, a Function Leader Character (FLC) needs to be used. This is a character that precedes a function name so that Z/XDC's parsing routine can know that what follows is supposed to be a function name. Example: **FORMAT PRIVATE~ASID(JES2).**

The FLC character is always permitted. In some contexts (such as above), it is required, in others, it is optional. Generally, the FLC is optional when the function name is not immediately preceded by alphanumeric characters. Example: **FORMAT R1?+X4(R5?).** Here, the X4 function is preceded by a plus sign, so Z/XDC can recognize the function name without requiring an FLC character. Nonetheless, the FLC character may be used whenever desired regardless of whether or not it is actually necessary. Example: **FORMAT R1?+~X4(R5?).**

The default FLC character is the tilde (~). The SET FLC command and the PROFILE command can be used to change this. The LIST FLC command and the PROFILE command both can be used to display what the FLC character currently is. See the following for more information.

HELP COMMANDS SET FLC
HELP COMMANDS LIST FLC
HELP COMMANDS PROFILE
HELP PROFILES

Example:

FORMAT 03BACD~ALET(00010003)
In this example, the tilde (~) is necessary to inform Z/XDC exactly where the hex address ends (03BACD) and the function name begins (ALET). Without the tilde Z/XDC would have no way of knowing how to parse the string "03BACDALET".

Help FUNctions ALet

The ALET(...) function changes the space being referenced by an address expression. The function's input is an Access List Entry Token (ALET). The new space may be an address space or a data space, depending upon the ALET.

The effect of the ALET(...) function is temporary. It lasts for the duration of the current Z/XDC command or until overridden by an ASID(...) or another ALET(...) function call in the same address expression.

The ALET(...) function interprets generic ALETs as follows:

X'00000000' - In most cases, this refers to the user program's retry level instance of the Primary Address Space. But when this ALET is obtained from an error level register (e.g. ALET(EAR5)), Z/XDC interprets it as referring to the **error level** instance of the Primary Address Space.

X'00000001' - In most cases, this refers to the user program's retry level instance of the Secondary Address Space. But when this ALET is obtained from an error level register (e.g. ALET(ER5)), Z/XDC interprets it as referring to the **error level** instance of the Secondary Address Space.

X'00000002' - This always refers to the Home Address Space. (There is no distinction between "error level" and "retry level" for the Home Address Space.)

For more information, see HELP VIRTMEM ALETS.

When in Local Address Space Mode (i.e. not in Foreign Address Space Mode), the ALET(...) function can be used to select any address space or data space having an entry either in the Home Address Space PS-AL or in the current task's DU-AL. (For more information, see HELP VIRTMEM ACCESSLISTS.)

Syntax:

```
ALET( alet#         )
      ARn
      Rn or RHn
      alet#-address
      0 or 1 or 2
      null
```

alet#
If the ALET(...) function's operand is a pure hexadecimal number that is exactly eight digits long, then Z/XDC assumes that the operand is the specific Access List Entry Token (ALET) for an accessible address space or data space. Z/XDC attempts to use the token for subsequent storage references.

ARn
If the ALET(...) function's operand is a name of an access register (AR5, 5AR, EAR0, etc.), then the value contained within that register is extracted and used as an ALET for subsequent storage references. The value in the associated general register is not referenced or used.

Rn or **RHn** or **ERn** or **ERHn**
If the ALET(...) function's operand is a name of the low half (R2, 0ER, etc.) or the high half (RH10, 14EHR, etc.) of a general register, then Z/XDC considers the value contained within that register to be an ALET, so Z/XDC extracts that value and uses it as an ALET for subsequent storage references. The value in the associated access register is not referenced or used.

alet#-address
If the given function operand neither is a pure 8-digit hexadecimal number nor is a register name, then Z/XDC attempts to evaluate the operand as an address expression. If successful, then Z/XDC extracts a 4-byte value from the targeted storage location for use as an ALET for subsequent storage references.

0 or **1** or **2**
Given or extracted generic ALET values of 0 and 1 refer to the user program's retry level Primary Space and Secondary Space, respectively. A specified or extracted value of 2 always refers to the Home Space.

When the ALET value is extracted from an error level register (access or general), the error level Primary and Secondary Spaces are used for values 0 and 1, respectively, Otherwise, the retry level Primary and Secondary Spaces are used for values 0 and 1.

For more information, see HELP VIRTMEM ALETS.

null
"ALET()" refers to the user program's Home Address Space.

USE4
USE4 tells the address expression analyzer to recall the ALET number that was saved by a prior SAVE4() function call within the same address expression.

This function is intended for internal use by the LIST VARIABLES command.

The ALET(...) function can appear anywhere in an address expression except within an offset term. It may be placed either ahead of the base term, following the base term, between offset terms, or following the last offset term. (See HELP ADDRESSING SYNTAX for information about the structure of address expressions.)

See HELP FUNCTIONS CHART for a table that summarizes the placement rules for all built-in functions.

Examples:

FORMAT R5?~ALET(AR3)
The address/data space designated by the ALET in AR3 is selected. The location in that space pointed to by general register R5 is displayed.

EQUATE MATRIXSP 0~ALET(00010003) 7FFFFFFF
FORMAT MATRIXSP+3BC000
This is an example of how a data space can be assigned a name in order to simplify subsequent references.

- The EQUATE command defines a symbol named MATRIXSP to represent the starting location (X'00000000') of a data space designated by ALET X'00010003'. (It does not matter whether or not the data space's actual starting address is zero.) For convenience, the equate is assigned a coverage length of X'7FFFFFFF'.
- The FORMAT command (and other commands, as well) can now reference locations within that data space by using the MATRIXSP name.

EQUATE MYSPACE 0~ALET(R5?ALET(AR3)) 7FFFFFFF
FORMAT MYSPACE+104C
The first of these commands ("EQUATE") defines an equate named MYSPACE to label the start of a data space whose ALET has been stored in another data space. The ALET for that other data space is in AR3. General register R5 contains the address (in the other data space) of the first data space's ALET. The second command (FORMAT) displays location X'0000104C' within the data space.

FORMAT 35B000~ALET(R5?)
FORMAT 35B000~ALET(AR5?)
FORMAT 35B000~ALET(EAR5)
These three commands appear similar, but their behaviors are all quite different:

FORMAT 35B000~ALET(R5?)
An ALET is obtained from the retry level instance of the Primary Address Space at the 4-byte location pointed to by the retry level instance of general register R5. (Access register AR5 is not referenced.) Location X'0035B000' of the address or data space designated by that ALET is displayed. If the ALET value is X'00000000' or X'00000001', then the address space displayed is the retry level instance of the Primary or Secondary Address Space, respectively.

FORMAT 35B000~ALET(AR5?)
An ALET is obtained from the space designated by the retry level instance of **access** register AR5 at the 4-byte location pointed to by the retry level instance of **general** register R5. Location X'0035B000' of the address space or data space designated by that ALET is displayed. If the ALET value is X'00000000' or X'00000001', then the address space displayed is the retry level instance of the Primary or Secondary Address Space, respectively.

FORMAT 35B000~ALET(EAR5)
An ALET is obtained directly from the error level instance of **access** register AR5. (General register R5 is not referenced.) Location X'0035B000' of the address space or data space designated by that ALET is displayed. If the ALET value is X'00000000' or X'00000001', then the address space displayed is the **error level** instance of the Primary or Secondary Address Space, respectively.

SET ASID JES2
...
FORMAT ASID()ALET(AR0)1000
- The first command sets Foreign Address Space Mode to JES2's address space.
- JES2 remains the default Target Address Space for subsequent Z/XDC commands.
- The ASID() function call in the FORMAT command temporarily overrides the current Target Address Space in order to access the cluster of address and data spaces that are accessible to the user program in the local Home Address Space.
- The ALET(AR0) function then selects the address or data space designated by the ALET in access register AR0. (This ALET must represent an entry either on the PS-AL for the user program's Home Address Space or on the DU-AL for the user program's current task.) Location X'00001000' of that space is then displayed.

Help FUNctions ASId

The ASID(...) function changes the space being referenced by an address expression. In effect, this function sets, changes, or terminates Foreign Address Space Mode (FASM mode) for only the duration of the current Z/XDC command or until overridden by another ASID(...) function call in the same address expression. (To change the **default** FASM/LASM setting, use the **SET ASID command.)**

The ASID(...) function's input parameter is an address space reference (referred to by Z/XDC as **aspaceref).** that can be an ASID number, a jobname, a keyword or any of several other things. The new space may be any accessible address space (security permitting), or it may be real storage (security permitting). (It may not be a data space.) The complete syntax for aspaceref parameters can be found at HELP COMMANDS SYNTAX ASIDS.

Whenever the ASID(...) function is used, the ALET associated with the current address expression is zeroed.

The ASID(...) function cannot be used unless Z/XDC is running authorized. Also, when this function is used, Z/XDC checks system security to see if the current user has permission to access the targeted job (or TSO session or system task). (See HELP SECURITY for more information.)

Syntax:

```
ASID( aspaceref )

Alias        - ASN
```

aspaceref
This parameter identifies the address space which the current address expression is to reference from this point onwards. Briefly, aspaceref can be any of the following:
- A jobname.
- A keyword indicating the home space or the retry level or error level, primary, secondary, or instruction execution address space. Examples: HOME, PASID, ESASID, IASID, etc.
- The keyword REAL, indicating real storage.
- An ASID number.

- An address expression pointing to an ASID number.
- A register containing an ASID number.
- The name of an equate or dsect that has been assigned to represent storage in an address space. (The assigned address space is used.)
- A null value indicating that the home address space is to be referenced.

For detailed information, see HELP COMMANDS SYNTAX ASIDS.

The ASID(...) function can appear anywhere in an address expression except within an offset term. It may be placed either ahead of the base term, following the base term, between offset terms, or following the last offset term. (See HELP ADDRESSING SYNTAX for information about the structure of address expressions.)

See HELP FUNCTIONS CHART for a table that summarizes the placement rules for all built-in functions.

Examples:

```
DMAP HASPDOC.HCT
USING HCT ASID(JES2)HASJES20.X#1
```

The first command reads a dsect map for the Hasp Control Table from a load module named HASPDOC. The second command assigns that dsect to represent the HCT control block located at the load point of the HASJES20 load module located in JES2's Home Address Space.

```
DMAP XDCMAPS.PSA
DMAP .PCCA
USING PSA 0
USING PCCA .PSAPCCAR?~ASID(REAL)
FORMAT PCCA
```

These commands do the following:

- The first command reads a dsect map for the System's Prefixed Storage Area from a load module named XDCMAPS.
- The second command reads a dsect map for the System's Physical Configuration Communication Area from the same load module.
- The third command assigns the PSA map to represent low storage.
- The fourth command assigns the PCCA map to represent, in real storage, the location of the current CPU's PCCA.
- The fifth command displays that PCCA via its real storage address. Notice that real storage is displayed without it having to be explicitly requested in the address expression (or via a prior SET ASID REAL command). This is because "real storage" has been made an attribute of the PCCA map by the USING command.

```
SET ASID VTAM
...
FORMAT ASID()R5?
```

- The first command sets Foreign Address Space Mode to VTAM's address space.
- VTAM remains the Target Address Space for subsequent Z/XDC commands.
- The FORMAT command temporarily overrides the Target Address Space in order to display the storage pointed to in the user program's Home Address Space by general register R5.

FORMAT 5000~ASID(1)
FORMAT 5000~ASID(0001)
FORMAT 5000~ASID(0001+0)
The first two of these commands display location X'00005000' in the System's Master Scheduler address space (ASID=1). The ASID(...) function's operand in the third command, on the other hand, is not a pure hexadecimal number, so Z/XDC evaluates it as an address expression. Since this expression resolves to location X'00000001', Z/XDC then attempts to load a 2-byte value from that location and use it as an ASID number (not a good idea).

EQUATE JES2ASID R3? D 2
FORMAT 5000~ASID(JES2ASID)
FORMAT 5000~ASID(JES2ASID+0)
Suppose that JES2's ASID number has been obtained from somewhere and stored locally at a location pointed to by general register R3. Suppose further that an EQUATE command has been issued to define a label named "JES2ASID" to represent the stored location of JES2's ASID number. Then:

- The first FORMAT command will fail to display JES2's storage. This is because the ASID(...) function's operand is given as a pure standard name, so instead of being evaluated as an address expression, Z/XDC instead attempts to interpret it as the name of a job, TSO session, or system task.
- In the second FORMAT command, the "JES2ASID" name is no longer pure. It is now modified by "+0", so now Z/XDC does evaluate it as an address expression, extracts JES2's actual ASID number that has previously been stored at the expression's resolved location, and thus displays location X'00005000' from JES2's Home Address Space.

FORMAT R5?~ASID(INIT)
This command attempts to display the location pointed to by R5 but in an initiator's address space. However, this command almost always will fail because usually more than one initiator exists in the System.

Help FUNctions ASN

The ASN(...) function is an alias of ASID(...). See HELP FUNCTIONS ASID for more information.

Help FUNctions F4

The F4(...) function is an alias of the S4(...) function. Please see **HELP FUNCTIONS S1-THRU-S8** for more information.

Help FUNctions H2

The H2(...) function is an alias of the S2(...) function. Please see **HELP FUNCTIONS S1-THRU-S8** for more information.

Help FUNctions ILC

The ILC(...) function returns the offset to the nth machine instruction past the location calculated so far in the current address expression.

Syntax:

```
ILC( number )
```

number
This is a decimal number indicating which following machine instruction is to be found. This number may be 0 or any positive value. If number is omitted then 1 is used by default.

The ILC(...) function can appear only within an offset term of an address expression. It may not appear between terms. (See HELP ADDRESSING SYNTAX OFFSETTERMS for more information.)

See HELP FUNCTIONS CHART for a table that summarizes the placement rules for all built-in functions.

The instruction pointed to by the address expression's location calculated so far is used as the ILC(...) function's starting location. This location is considered to be the 0th instruction; therefore, ILC(0) always returns an offset of 0.

ILC(1) causes the function to return the offset of the first instruction following the function's starting location. That offset, of course, depends upon the length of the instruction found at the starting location.

Processing of the ILC(...) function requires Z/XDC to scan storage forward from the starting location. Dead traps that might be encountered by the scan are considered to be single instructions. (See HELP BREAKPOINTS DEADTRAPS for an explanation of DEAD traps.)

If the scan encounters invalid opcodes, then the ILC(...) function presumes an instruction length based upon the invalid opcode's two hi-order bits:
B'00' - 2-byte instruction.
B'01' - 4-byte instruction.
B'10' - 4-byte instruction.
B'11' - 6-byte instruction.

If the scan encounters inaccessible storage, then the ILC(...) function issues an error message and aborts.

If an extremely large value is given for number (example: "ILC(999999)"), ILC(...)'s scan may take a long time. If you find that the scan is taking too long, you can use an attention signal to abort the command. (See HELP ATTENTIONS INXDC for information about interrupting long running Z/XDC commands.)

NXSEQ(...) is a related function that performs similarly to ILC(...). See HELP FUNCTIONS NXSEQ for more information.

Examples:

GO R14?+ILC()
GO R14?+ILC(1)
These two commands are equivalent. Assuming that R14 points to a return address (e.g. the **first** instruction past a BAL or BALR), this command causes user program execution to skip that first instruction and resume at the second instruction past the BAL or BALR. This works regardless of the length of that first instruction.

Note, the NXSEQ(...) function offers a similar but different method for accomplishing the same thing. See HELP FUNCTIONS NXSEQ for more information.

FORMAT R14?
TRAP +0 +ILC(1) +ILC(2) +ILC(3)
The FORMAT command displays program code (presumedly) starting at the location pointed to by R14. More importantly, it also sets Z/XDC's Current Display Pointer ("CDP") to point to the start of the displayed location. (See HELP ADDRESSING IMPLICIT for more information about the CDP).

The TRAP command then makes use of the CDP to set a family of four breakpoints onto the first four machine instructions starting at the location pointed to by R14. (See HELP BREAKPOINTS FAMILIES for information about families of breakpoints.)

Help FUNctions IEq

The IEQ(...) function inserts the value of an IPCS equate at this point in the current address expression.

Note that this function can only be used under dump/XDC.

Syntax:

IEQ(equatename)

equatename
This is the name of the IPCS equate whose value is to be inserted. If the equate doesn't exist (including the case where the name is defined as an IPCS literal rather than an IPCS equate) an error occurs.

The IEQ(...) function can only be used at the start of an address expression.

When using dump/XDC, this function can be used at the start of a Z/XDC address

expression to start it with the value of an IPCS equate. The IPCS equate:
- could be one created 'under the covers' by IPCS, for example ASCB5.
- Can be for a value:
  - That is a virtual address (in common or any address space).
  - That is an absolute (real) address.
  - That is in a dataspace. Note that in this case, dump/XDC will ensure that an ALET that allows this dataspace to be accessed at any time is also created.
- May be differently named to the one used. For example, while IPCS uses (and displays) ASCB5, internally the actual equate name is ASCB00005. dump/XDC understands this, and automatically uses the correct name when calling IPCS.

Regardless of what the IPCS equate is for, the address, ASID, and ALET for the address expression are all set appropriately.

See HELP FUNCTIONS CHART for a table that summarizes the placement rules for all built-in functions.

Examples:

D IEQ(ECVT)
This example displays the ECVT in hex and character. The address of the ECVT is in an IPCS equate named **ECVT.**

EQUATE MYRCE IEQ(RCE)+0 0580
LIST EQUATES MYRCE
D MYRCE
This example first creates a Z/XDC equate called **MYRCE** using an address expression that uses the IPCS equate **RCE** with the addition of 0 to it, and a length from the current RCE definition.
The Z/XDC equate is then displayed.
Finally, the storage associated with the created Z/XDC equate is then displayed.

Help FUNctions ILIt

The ILIT(...) function inserts the value of an IPCS Literal equate at this point in the current address expression.

Note that this function can only be used under dump/XDC.

Syntax:

ILIT(literalname)

literalname
This is the name of the IPCS literal whose value is to be inserted. If the literal equate doesn't exist (including the case where the name is defined as an IPCS symbol but not literal) an error occurs.

The ILIT(...) function can be used anywhere within an address expression.

See HELP FUNCTIONS CHART for a table that summarizes the placement rules for all built-in functions.

Help FUNctions INdirect

The **INDIRECT(...)** function can be used in an address expression anywhere that an indirect operator (% ? !) can be used. It performs the same indirect addressing operations as do the normal indirect operators, plus several additional ones. Essentially, it solves the problem of trying to support more than three different kinds of indirect operations with only three distinct indirect operator characters.

Syntax:

```
INDIRECT( 24     ZEROOK  )
          31     ZERONOK
          32     omitted
          64
          !
          AMODE
          4BYTE
          HLRn
          H0Rn
          mask
```

Aliases:

- **%** is an alias of **24.**
- **?** is an alias of **31.**
- **CURRENT** is an alias of **AMODE.**
- **0OK** is an alias of **ZEROOK.**
- **0NOK** is an alias of **ZERONOK.**

- **!** is **not** an alias of **64.** (See **HELP COMMANDS SET BANG** for the meaning of **!.)**

Rules:

- The **INDIRECT(...)** function accepts one or two operands.
- When both operands are given, they must be given in the order shown.
- The second operand values, when present, may be used in any combination with the first operand values.

Operands:

%
24
Performs a 24-bit wide indirect operation. The new address is loaded from the lo-order 3 bytes of the fullword pointed to by the current point in the current address expression calculation. The fullword need not be aligned.

?
31
Performs a 31-bit wide indirect operation. The new address is loaded from the lo-order 31 bits of the fullword pointed to by the current point in the current address expression calculation. The fullword need not be aligned.

32
Performs a 32-bit wide indirect operation. The new address is loaded from the **entire** 32 bits of the fullword pointed to by the current point in the current address expression calculation. The fullword need not be aligned. (This makes it possible to load pointers into the **2G-4G** section of storage. Formerly, this was the **dead zone,** but now it is the **Java work area.)**

64
Performs a 64-bit wide indirect operation. The new address is loaded from the entire 64 bits of the doubleword pointed to by the current point in the current address expression calculation. The doubleword need not be aligned.

!

Performs the type of indirect operation that is currently established by the **SET BANG** setting:

- When SET BANG **AMODE** is in effect, **!** performs either a 24-bit indirect or a 31-bit indirect or a 64-bit indirect according to the addressing mode of the retry level or error level PSW.

  The error level PSW will be used when the current address expression is based upon either the EPSW or any error level register (ER5, for example). Otherwise, the addressing mode of the retry level PSW will be used.

- When SET BANG **64BIT** is in effect, **!** performs a 64-bit indirect regardless of the addressing mode of any PSW.

  The factory default setting for **BANG** is **64BIT.**

AMODE
CURRENT
Performs a 24-bit wide, 31-bit wide, or 64-bit wide indirect operation according to the retry level or error level environment's current AMODE. The error level environment is used if the address expression within which this function occurs is

based upon an error level PSW (EPSW?...) or register (EAR5?..., for example). Otherwise, the retry level environment is used.

4BYTE
Performs only a 24-bit or 31-bit wide indirect operation according to the retry level or error level environment's current AMODE. If the current AMODE is 64, then only a 31-bit wide indirect operation is performed.

HLRn
An indirect operation is performed in a way that is similar to that which is performed by the **BAKR** machine instruction (in its R-one operand) and is reported by the **BSM** and **BASSM** machine instructions (in their R-one operands): A 24-bit, 31-bit or 64-bit indirect operation is performed according to the high and low bits of the **32-bit** general register indicated by **n** from the given **HLRn** operand:
- If the register's **lo-order** bit is on, then a 64-bit indirect operation is performed.
- Otherwise, if the register's **hi-order** bit is on, then a 31-bit indirect operation is performed.
- Otherwise, a 24-bit indirect operation is performed.

Notice! The registers indicated by the **HLRn** operands are the **lo-order halves** of the 64-bit general registers. Therefore:
- The **lo-order** bit of register **Rn** is actually bit number **63** of register **RWn.**
- The **hi-order** bit of register **Rn** is actually bit number **32** of register **RWn.**

Notice! All the rest of the bits in register Rn are ignored! The indirect action is **not** performed against the address in Rn. Instead, it is performed against the **address calculated so far** [just like for any other operand of the INDIRECT() function].

H0Rn
An indirect operation is performed in a way that is similar to that which is performed by the **PT** and **PTI** machine instructions (from their R-two operands):
- If the current **PSW** is in 64-bit AMODE, then a 64-bit indirect operation is performed.
- Otherwise, if the 32-bit **Rn** register indicated by the given **H0Rn** operand has its **hi-order** bit on, then a 31-bit indirect operation is performed.
- Otherwise, a 24-bit indirect operation is performed.

Notice! The registers indicated by the **H0Rn** operands are the **lo-order halves** of the 64-bit general registers. Therefore:
- The **hi-order** bit of register **Rn** is actually bit number **32** of register **RWn.**

Notice! All the rest of the bits in register Rn are ignored! The indirect action is **not** performed against the address in Rn. Instead, it is performed against the **address calculated so far** [just like for any other operand of the INDIRECT() function].

mask

This defines the indirect operation in terms of a mask:
- The mask must be a hexadecimal string from 1 to 16 digits long.
- The string may contain one or more underscores (_) at any point. (They are permitted for human readability.) All underscores are ignored.
- The number of digits in the string defines the width of the pointer field to be loaded from storage (1 to 8 bytes, with zero padding on the left if an odd number of digits are given).
- The mask's bits define which data from that field constitute the pointer and which bits are to be zeroed.

Examples:

~INDIRECT(FFFFFF)
This loads a 24-bit wide pointer from a 3-byte wide field.

~INDIRECT(00FFFFFF,0OK)
This loads a 24-bit wide pointer from a 4-byte wide field. A zero'd value is OK.

~INDIRECT(7FFFFFFF)
This loads a 31-bit wide pointer from a 4-byte wide field.

~INDIRECT(FFFFFFFF_FFFFFFFF)
This loads a 64-bit wide pointer from an 8-byte wide field.

~INDIRECT(0000FF0000)
This loads a 5-byte wide pointer using only the third byte of a 5-byte wide field. (All the other bits are zero'd.)

ZERONOK
0NOK
omitted
If the indirect operation loads a **zero value,** then an error **(DBC047E)** is reported, and the current command is aborted. Note, this is the default action for **all** indirect operators: **% ? !** and the **RDDD()** and **INDIRECT()** built-in functions. For more information, see **HELP ADDRESSING SYNTAX BETWEENTERMS INDIRECT.**

ZEROOK
0OK
omitted
The loading of a zero value by the indirect operation **is permitted.** Note, This is the only method supported by Z/XDC that permits this.

Examples:

EQ @R0 @REGS+0*4~INDIRECT(4BYTE 0OK) FLOAT 1000
This creates an equate named **@R0** and assigns it to represent the location pointed to by retry level general register R0. Notes:

- The **0OK** operand means that the creation of the **@R0** equate will succeed even if the R0 slot at the **@REGS** location contains zeros. (For more information about the **@REGS** built-in equate, see **HELP EQUATES BUILTIN REGSETS.)**

- The **4BYTE** operand means that if the retry level environment's AMODE is 24, then only the lo-order 24 bits of register R0 are used for resolving the indirect pointer. However, if the AMODE is 31 or 64, then the lo-order 31-bits of the register are used.

FORMAT CRW1SAVE~INDIRECT(FFFFFFFF_FFFFF000)~ASID(REAL)
The hardware's control register 1 is defined to contain a pointer to the primary address space's top segment table entry or region table entry. The pointer is defined to be the equivalent of all 64 bits of the register but with the lo-order 12 bits zeroed.

Suppose that a copy of control register 1 (all 64 bits) has been saved in a field named CRW1SAVE. Then the above FORMAT command would display (in real storage) the segment/region table entry origin being pointed to by CRW1SAVE.

ZAP RW2=00000002,33444444
FORMAT RW2~INDIRECT(HLR1)
These commands cause a location to be displayed that is pointed to by RW2. The interpretation of the pointer in RW2 is controlled (as follows) by the hi- and lo-order bits in R1:

- If R1 contains (for example) X'98887775' (ie, has its lo-order bit on, the hi-order bit does not matter), then the address contained within RW2 (ie, the **address calculated so far)** will be interpreted as a **64-bit address,** and so the location displayed will be X'00000002_B3444444'.

- If R1 contains (for example) X'98887774' (ie, has its lo-order bit off and its hi-order bit on), then the address contained within RW2 will be interpreted as a **31-bit address,** and so the location displayed will be X'00000000_33444444'.

- If R1 contains (for example) X'18887774' (ie, has both its hi-order and lo-order bits off), then the address contained within RW2 will be interpreted as a **24-bit address,** and so the location displayed will be X'00000000_00444444'.

Help FUNctions Lmod

The LMOD(...) function returns the entry point (address and address space) of a named load module (or program object).

While address expressions allow load module names (including program object names) to be used directly, and you can quote the first part of an address expression if it contains characters that are not allowed by Z/XDC in the first part of an address expression unless quoted, you can use the LMOD(...) function to ensure that the value is only used as a load module name.

The syntax of a name (etc.) that Z/XDC supports is explained in **HELP COMMANDS SYNTAX**

The **LMOD(...)** function allows you to specify **any** load module or program object name regardless of whether or not the name contains characters that Z/XDC would not normally allow.

See HELP FUNCTIONS CHART for a table that summarizes the placement rules for all built-in functions.

Syntax:

```
LMOD( [opening-quote]name[closing-quote] )
```

opening-quote
The opening quote is optional. If it is not specified then the load module (etc.) name starts after the opening parenthesis following the **LMOD** function name.

If it is specified, it can be one of the following characters:
- Apostrophe (')
- Literary quote(")

Note that the presence or absence of an opening quote affects how the load module (etc.) name is matched. See below for full details.

name
The load module or program object member name or USS program object file name or USS program object path name

The maximum length of this name is 64 characters.

closing-quote
If an opening quoter has been specified, the same character (followed by the function's closing parenthesis) is required to delimit the end of the name.

If no opening quote has been specified, then no closing quote can be specified.

Search Order

The **LMOD(...)** function searches for a matching module name using the following rules...

- only programs that are actually in storage are considered.
- The search uses the standard z/OS 'search order'. That is: - the Job Pack Queue - Any ancestor Job Pack Queues - the DLPA - The FLPA - the MLPA - The PLPA
- the first program found that matches the supplied name (using the rules documented below) is used to calculate the return value.

Name Match Rules

The name of a program that has been located using the above search rules is matched against the provided name (referred to as 'value' below) using the following rules:

- First, the provided value is analyzed to determine if is a valid PDS member name. The rules are:
  - the length must be 1 to 8.
  - all characters must be alphabetic ('a'-'z' or 'A'-'Z') or numeric ('0'-'9') or 'national' ('@', '#', '$')
  - the first character must not be a digit

  If it is a valid PDS member name, this is flagged and an upcased copy of the value is saved.

- If the value was not quoted:
  - If the value is flagged as being a PDS member name a scan of all the programs in the search order is done and those programs that do not have a path name specified are compared against the PDS member name value.
    If a match is found the search stops.
  - a scan of all the programs in the search order is done and those programs that have a path name specified are compared against the full value using the rules documented below. (Note that this will always be a file name because the value can only have a '/' in it if quoted).
    If a match is found the search stops.

- If the value was quoted:
  - a scan of all the programs in the search order is done and those programs that have a path name specified are compared against the full value using the rules documented below.
    If a match is found the search stops.
  - If the value is flagged as being a PDS member name a scan of all the programs in the search order is done and those programs that do not have a path name specified are compared against the PDS member name value.
    If a match is found the search stops.

(Note that the above rules mean that if a provided value matches both a program with a PDS member name and a program with a path name the non-quoted value will be treated as matching the PDS member name whereas the quoted value will be treated as matching the path name)

File or path name match?

When a program has been loaded from an HFS file using the USS facilities (i.e., the program has a path name), a match is done on the file name or the path name as follows:

- if the supplied value contains any '/' (forward slash or solidus) characters, the program's full path name must match the supplied value exactly.
- if the supplied value does not contain any '/' (forward slash or solidus) characters, the program's file name (i.e., the part of the program's path name after the last '/') must match the supplied value exactly.

Examples:

FORMAT LMOD(IEFBR14)
This produces a display of the instructions of the IEFBR14 load module. This example has no quotes specified, and shows that the LMOD(...) function can be used in cases where it may not actually be needed. (It is assumed that there is no USS-loaded

program with a file name of 'IEFBR14')

FORMAT LMOD('IEFBR14')
This performs identically to the preceding example and shows the use of quotes.

FORMAT LMOD(fred)
This produces a display of the instructions in either the 'FRED' load module (or program object) or a USS-loaded program loaded from a program object file where the file name is 'fred'. If both names exist, the 'FRED' load module (etc.) will take precedence over any USS-loaded program with a file name of 'fred'.

FORMAT LMOD('fred')
This produces a display of the instructions in either the 'FRED' load module (or program object) or a USS-loaded program loaded from a program object file where the file name is 'fred'. If both names exist, the USS-loaded program with a file name of 'fred' will take precedence over the 'FRED' load module.

FORMAT LMOD(|this program name has embedded blanks|)
This example shows how quotes can be used to provide a name that has embedded blanks.

Note that the match is only on the file name.

FORMAT LMOD('this/program/name/has/a path name')
This example shows how a path name can be provided.

FORMAT LMOD('this name has a '' in it')
This example shows how a quote itself can be specified in a name.

Help FUNctions NXInst

The NXINST(...) function returns the location (address and address space) of the machine instruction that will be executed **next after** the current instruction pointed to by the retry level PSW. The function takes into account:
- Whether or not the current instruction is a branching instruction.
- If it is a branching instruction, then whether or not a branch will, in fact, occur.
- Whether or not the current instruction will cause a space switch.

This function is particularly useful in conditional breakpoints for tracking down bugs involving wild branches. See examples given below, and see HELP BREAKPOINTS CONDITIONAL for more information.

The NXINST(...) function sets both an address expression's address and its address space. Accordingly, it can be used only as an address expression's base term. See HELP ADDRESSING SYNTAX BASETERM for more information.

See HELP FUNCTIONS CHART for a table that summarizes the placement rules for all built-in functions.

Syntax:

```
NXINST( omitted )
```

PC
SVC

omitted vs. **PC** vs. **SVC**
In most cases the presence or absence of an operand makes no difference. A difference occurs **only** when the retry level PSW is currently pointing to a PC or SVC or TRAP2 or TRAP4 instruction. In this case:

- When NXINST() is used, the NXINST function returns the location of the instruction following the PC or SVC or TRAP2 or TRAP4 instruction.

- When either NXINST(PC) or NXINST(SVC) is used (and the retry level PSW is pointing to a PC or SVC or TRAP2 or TRAP4 instruction), the NXINST function returns the location of the first instruction of the PC or SVC service routine or TRAP intercept routine that the PC or SVC or TRAP2 or TRAP4 instruction will cause to be executed.

Notice that the "PC" and "SVC" operands are actually aliases of each other. Each works equally well for the PC, SVC, TRAP2, and TRAP4 instructions.

Examples:

FORMAT NXINST()
This produces a display of the instruction that will be executed next after the current instruction pointed to by the retry level PSW. The function takes into account whether the current instruction will or will not cause a branch and whether it will or will not cause a space switch (as can happen with the PT and PR instructions).

FORMAT NXINST(PC)
This performs identically to the preceding example except when the retry level PSW points to a PC or SVC or TRAP2 or TRAP4 instruction (or an EX thereof). In this case, the preceding example would cause the instruction following the PC, SVC, TRAP2, or TRAP4 instruction to be displayed; whereas, this example would cause the PC, SVC, TRAP2, or TRAP4 service routine’s first instruction to be displayed.

TRAP NXINST()
This causes a transient breakpoint to be placed onto the machine instruction that will be executed following the current instruction. (When the current instruction is a PC, SVC, TRAP2, or TRAP4 instruction, the breakpoint will be placed onto the instruction following that instruction, rather than into the service routine itself.)

TRAP NXINST(PC)
This performs identically to the preceding example except when the retry level PSW points to a PC, SVC, TRAP2, or TRAP4 instruction (or an EX thereof). In this case, Z/XDC is being instructed to place a breakpoint onto the PC, SVC, TRAP2, or TRAP4 service routine’s first instruction. However, doing so would always cause the debugging session to fail, and sometimes it would also cause other address spaces

and even the entire z/OS Operating System itself to fail as well. Therefore, Z/XDC considers this to be a special case and so refuses to set the breakpoint. Instead, it issues message DBC141E and aborts the command.

In order for Z/XDC to recognize this special case **all** of the following conditions have to be met:

- The command must be AT, ATX, or TRAP.
- An operand of the command must be exactly "NXINST(PC)" or "NXINST(SVC)". In other words the address expression must be a pure NXINST function with the PC or SVC operand given.
- The machine instruction currently pointed to by the retry level PSW must be a PC, SVC, TRAP2, or TRAP4 instruction.

When all of these conditions are met, Z/XDC refuses to set the requested breakpoint and aborts with message DBC141E.

TRAP NXINST(PC)+0

This example bypasses the special case described above. Because the address expression no longer is a pure NXINST function call, Z/XDC does not recognize the special danger created by setting the breakpoint as directed. So, unless prevented by authority level or security rules, Z/XDC will set the breakpoint, and depending upon the PC, SVC, TRAP2, or TRAP4 involved, your system may crash! Setting a TRAP2-style breakpoint onto the target of a TRAP2 or TRAP4 instruction will cause an infinite spin loop, because the TRAP2-style breakpoint will branch to itself (assuming the TRAP hardware is installed and the program is set up to use the TRAP facility). (Obviously, Z/XDC cannot set the breakpoint into protected storage when the debugging session is not running authorized. Also, a customer can use Z/XDC's security rules to help protect against this problem. For information about security rules, see HELP SECURITY.)

Regarding setting breakpoints into dangerous places: The number of and characteristics of such places are far too vast and far to variable(!) for Z/XDC to make any serious attempt to recognize and protect against. For example, setting a breakpoint into a dangerous location may be "OK" one time but "bad" another, depending upon the Z/XDC user's skill level and the steps that he may or may not have taken to secure against interfering activity during his debugging session. The best that Z/XDC can do is try to recognize a few of the more error prone situations (a judgment call!) and provide at least some warning. Customers that need more comprehensive protection should investigate using Z/XDC's System Security rules to control who can use Z/XDC authorized and what areas of storage they can and cannot zap.

TRACE BY (XADDR(NXINST()-XADDR(+0)&FFF),EQ,05104000)

This is a conditional trace that allows execution to proceed until a branching instruction is reached that will jump to any location within the page whose virtual address is X'5104000'. For more information about conditional tracing, see HELP BREAKPOINTS CONDITIONAL.

TRACE BY (XADDR(NXINST()),AND,7FFFF000,EQ,05104000)

This accomplishes the identical result as the prior example. In the prior example, the clearing of the three lo-order nibbles was done within the address expression portion ("NXINST()-XADDR(+0)&FFF") of the command. In this example the clearing is

done via the mask portion ("7FFFF000") of the conditional expression.

Note, there is a problem with both of the above two examples: If execution reaches the page via fall thru from the preceding page, then execution doesn't stop until it reaches a branching instruction that will branch to elsewhere within that page. If no such instruction is reached, then execution does not stop.

TRAP 5104000
TRACE BY (XADDR(NXINST()),AND,7FFFF000,EQ,05104000)
This sequence solves the problem noted above.

TRAP 5104528
TRACE BY (XADDR(NXINST()-05104528),LT,0000DF27)
This sequence traps program execution at any branch (or fall thru) that will cause execution to enter the range of storage between X'5104528' and X'511244E' (inclusive). (Note that "LT" is a **logical** compare, and so a negative value compares high against a positive value.)

Help FUNctions NXSeq

The NXSEQ(...) function returns the location (address and address space) of the nth machine instruction past a given address.

This function sets both an address expression's address and its address space. Accordingly, it can be used only as an address expression's base term. See HELP ADDRESSING SYNTAX BASETERM for more information.

See HELP FUNCTIONS CHART for a table that summarizes the placement rules for all built-in functions.

Syntax:

```
NXSEQ(addressexpression,number)
NXSEQ(number,addressexpression)
```

Note: The two operands may be given in any order. Also, either or both may be omitted. If both operands are given, and if both consist entirely of decimal digits, then Z/XDC considers the first operand to be the **number** operand and the second operand to be the **addressexpression** operand.

addressexpression
This is an address expression giving the location from which the nth following machine instruction is to be found. If omitted, then the default starting location is "PSW?", i.e. the address (and address space) pointed to by the retry level PSW.

number
This is a decimal number indicating which following machine instruction is to be found. This number may be 0 or any positive value. If number is omitted then 1 is used by default.

The instruction pointed to or defaulted to by the function's addressexpression operand (i.e. the "starting" instruction) is considered to be the 0th instruction. Accordingly, a value of 0 (for the number operand) causes the function to return the location of the starting instruction.

A value of 1 causes the function to return the location of the first instruction following the starting instruction. That address, of course, depends upon the length of the starting instruction.

Processing of the NXSEQ(...) function requires Z/XDC to scan storage forward from the starting location. Dead traps that might be encountered by the scan are considered to be single instructions. (See HELP BREAKPOINTS DEADTRAPS for an explanation of DEAD traps.)

If the scan encounters invalid opcodes, then the NXSEQ(...) function presumes an instruction length based upon the invalid opcode's two hi-order bits:

B'00' - 2-byte instruction.
B'01' - 4-byte instruction.
B'10' - 4-byte instruction.
B'11' - 6-byte instruction.

If the scan encounters inaccessible storage, then the NXSEQ(...) function issues an error message and aborts.

If an extremely large value is given for number (e.g. "NXSEQ(999999)"), NXSEQ(...)'s scan may take a long time. If you find that the scan is taking too long, you can use an attention signal to abort the command. (See HELP ATTENTIONS INXDC for information about interrupting long running Z/XDC commands.)

ILC(...) is a related function that performs similarly to NXSEQ(...). See HELP FUNCTIONS ILC for more information.

Examples:

GO NXSEQ(R14?)
GO NXSEQ(,R14?)
GO NXSEQ(R14?,1)
GO NXSEQ(1,R14?)
These four commands are equivalent. Assuming that R14 points to a return address (e.g. the **first** instruction past a BAL or BALR), this command causes user program execution to skip that first instruction and resume at the second instruction past the BAL or BALR. This works regardless of the length of that first instruction.

Note, the ILC(...) function offers a similar but different method for accomplishing the same thing. See HELP FUNCTIONS ILC for more information.

TRAP NXSEQ() NXSEQ(2) NXSEQ(3) NXSEQ(4) NXSEQ(5) NXSEQ(6)
GO
This is a series of commands that would be useful when program execution has reached a BAL or BALR (or BAS, BASR, or BASSM) going to a subroutine and you want to let the subroutine execute and then have Z/XDC recapture execution when the subroutine completes. A problem occurs if the subroutine is designed to make a vectored return

to any one of up to, say, six instructions following the BALR (or whatever) and you don't know which one it will be.

These commands cause a family of breakpoints to be placed on each of up to six possible return points. It then causes the user program to resume execution. The user program then calls the subroutine, the subroutine executes and returns, one of the six breakpoints is hit, and so Z/XDC recaptures execution. Z/XDC then clears all six breakpoints, displays the result, and awaits further commands. More specifically, these commands work as follows:

TRAP NXSEQ() NXSEQ(2) NXSEQ(3) NXSEQ(4) NXSEQ(5) NXSEQ(6)
This command causes a family of six transient breakpoints to be placed on the 1st through 5th machine instructions **following** the instruction pointed to by "PSW?" (the retry level PSW). (See HELP BREAKPOINTS FAMILIES for information about families of breakpoints.)

GO
This causes user program execution to resume. When the subroutine completes and execution reaches one of the six breakpoints, Z/XDC will again receive control.

Note, the above command sequence makes a particularly useful PF key. In fact, the factory default setting for F11 is this sequence. See HELP BREAKPOINTS TRACING PFKEYS for more information.

Help FUNctions Pclocate

The PCLOCATE(...) function returns the location (both ASID and virtual address) of the PC routine that would be called by a given PC number.

Syntax:

PCLOCATE(addr-exp)

Operands:

addr-exp
This can be any arbitrary address expression, but it is not used to reference storage. Instead, after resolution its lo-order 20 or 32 bits are interpreted as a PC number that Z/XDC then further resolves to the location of the PC routine that the PC number would cause to be called.

Comments

This function can be used regardless of whether or not Z/XDC is running authorized.

But if the PC routine is located in a different address space, then in order to **display** its storage, you will have to be both running authorized and permitted by System Security to do so.

The PCLOCATE(...) function can only be used as a base term. It may not be used anywhere else in an address expression. (See HELP ADDRESSING SYNTAX BASETERM for more information.)

See HELP FUNCTIONS CHART for a table that summarizes the placement rules for all built-in functions.

Examples:

FORMAT PCLOCATE(R14?)
When R14 contains a valid PC number, this command displays the location of the PC routine that would be called by that number.

FORMAT PCLOCATE(30F)
FORMAT PCLOCATE(10?+304?+0B0?)
Both of these commands display the PC routine that IBM's **ESTAEX** macro invokes to create or delete ESTAEX routines.

Help FUNctions RDdd

The RDDD(...) function extracts a machine instruction's s-con (base/displacement field), resolves the s-con to an address using a specified set of registers, and substitutes that address for the current address expression's "location calculated so far". In other words, the RDDD(...) function operates like an indirect operator.

If the base part of the extracted s-con is register R0, then the contents of register R0 are ignored, and the value zero is used instead as the base.

- The s-con to be extracted is extracted from the storage location pointed to by the address expression's "location calculated so far".
- The register set to be used in the conversion of the s-con to an address value is given by the function's operand.
- The width of the new address (24 bits, 31 bits or 64 bits) is governed by a required indirect operator (%, ? or !) that must follow the RDDD(...) function. Example: **FORMAT PSW?+2~RDDD(R)!**

For RX-type machine instructions, the RDDD(...) function does **not** consider index registers when extracting and resolving an s-con. It only looks at base registers. Examples:

L 15,12(3,5)
The hexadecimal translation of this instruction is X'58F3500C'. The s-con in this instruction is X'500C'. Therefore, the RDDD(...) function, when directed to resolve this s-con, adds X'00C' to the contents of general register R5, ignoring general register R3.

L 15,12(5)

The hexadecimal translation of this instruction is X'58F5000C'. The s-con in this instruction is X'000C'. Therefore, the RDDD(...) function, when directed to resolve this s-con, adds X'00C' to the value zero, ignoring general register R5 (and also ignoring the contents of general register R0).

L 15,12(,5)
The hexadecimal translation of this instruction is X'58F0500C'. The s-con in this instruction is X'500C'. Therefore, the RDDD(...) function, when directed to resolve this s-con, adds X'00C' to the contents of general register R5.

The RDDD(...) function **must** be followed by an indirect operator (% ? or !) to indicate whether the s-con's resolved address is to be truncated to 24 bits or to 31 bits.

Syntax:

```
RDDD( register-set-name ) %
      null                ?
                          !
```

register-set-name
An s-con consists of a base register number and a 12-bit displacement value. In order to resolve the s-con, Z/XDC has to add the displacement value to the contents of the specified register. However, Z/XDC has access to several register sets, so Z/XDC has to be told which register set it should use for resolving the s-con. The following register set names are recognized:

- **R** - Retry level general registers.
- **ER** - Error level general registers.
- **AR** - Retry level access registers.
- **EAR** - Error level access registers.

When an access register set is specified, Z/XDC uses the ALET found in the appropriate access register to select the address space or data space to which the resolved address value is applied. It then uses the corresponding general register set to resolve the s-con's address value to the correct location within that space.

null
RDDD() is equivalent to RDDD(R). It causes Z/XDC to resolve the targeted s-con using the retry level general registers.

indirect-operator
The RDDD(...) function **must** be followed by an indirect operator (% ? or !) to indicate whether the s-con's resolved address (which initially is resolved to 64 bits) is to be truncated to 24 bits or to 31 bits or left untruncated at 64 bits:

- **%** - The s-con's resolved address is truncated to 24 bits.
- **?** - The s-con's resolved address is truncated to 31 bits.
- **!** - The s-con's resolved address is truncated or not according to the current SET BANG setting:
  - **SET BANG 64BIT:** The resolved address remains untruncated at 64 bits wide.
  - **SET BANG AMODE:** The s-con's resolved address either remains untruncated at 64 bits wide, or is truncated either to 24 bits or to 31 bits. This choice is governed by the AMODE flags of either the retry level PSW **(PSW)** or the error level PSW **(EPSW),** depending upon whether the RDDD(...) function's operand references a retry level register set or an error level register set.

The RDDD(...) function can appear anywhere in an address expression that an indirect operator can appear. It may be placed either following the base term, between offset terms, or following the last offset term. It may **not** be placed either ahead of the base term or within an offset term. (See HELP ADDRESSING SYNTAX for information about the structure of address expressions.)

See HELP FUNCTIONS CHART for a table that summarizes the placement rules for all built-in functions.

Examples:

FORMAT PSW?+2~RDDD(R)%+20
Assume that the retry level PSW points to a **MVC 24(32,2),55(15)** instruction in the retry level Primary Space.
- The hexadecimal translation of this instruction is X'D2162018F037'.
- This instruction has two s-con's. They are X'2018' and X'F037'.
- The **"PSW?+2"** portion of the FORMAT command's address expression causes the expression's "location calculated so far" to be pointed to the first of these s-cons: X'2018'.
- The **"RDDD(R)"** function evaluates this s-con using retry level general register R2. The new address is the sum of X'018' added to the contents of R2.
- The **"%"** indirect operator causes Z/XDC to truncate this sum to 24-bits and then make it the expression's new "location calculated so far".
- The **"+20"** causes Z/XDC to add 32 (decimal) to the address expression's location calculated so far. This then becomes the storage location that is formatted and displayed.
- The storage location is in the retry level Primary Address Space, unless the PSW designates Home ASC-MODE (which would cause the displayed storage location to be from the Home Space).

FORMAT PSW?+2~RDDD(AR)%+20
This is the same as the prior example, except that **access** register AR2 is used used **after** the extraction and resolution of the s-con to designate a new address/data space for the remainder of the address expression. Note that **general** register R2 is still used for resolving the s-con's value. The extraction of the s-con is from the retry level Primary Space, unless the PSW designates Home ASC-MODE (which means the s-con is extracted from the Home Space).

Help FUNctions REAl

The REAL(...) function returns the real storage location of a given virtual location.

The REAL(...) function cannot be used unless Z/XDC is running authorized. Also, when this function is used, Z/XDC checks system security to see if the current user has permission to access real storage. (See HELP SECURITY for more information.)

Syntax:

```
REAL( addr-exp )
```

addr-exp
This can be any arbitrary address expression resolving to any location in any virtual address space or data space.

If addr-exp itself resolves to a real storage location, then the REAL(...) function basically NOPs: It just returns the same real storage location to which the expression resolved.

The REAL(...) function can only be used as a base term. It may not be used anywhere else in an address expression. (See HELP ADDRESSING SYNTAX BASETERM for more information.)

See HELP FUNCTIONS CHART for a table that summarizes the placement rules for all built-in functions.

Note, the **ASID(REAL)** function offers a different means by which real addresses can be accessed. Also note that the ASID(REAL) function can be used **anywhere** within an address expression that any other ASID(...) function can be used. See HELP FUNCTIONS ASID for more information.)

WARNING! Real storage locations are volatile. In most cases the real storage location of an object residing in virtual storage can change frequently and unpredictably. In fact such locations can change even during Z/XDC's processing of the REAL(...) function itself! Only the following storage locations can be relied upon to be stable for any length of time:
- Common storage pages that have been long term page-fixed.
- Private area pages that have been long term page-fixed **and** that are owned by **nonswappable** jobs or system tasks. (Page-fixed pages located within swappable address spaces can and do change real storage locations when the address space is swapped out and then swapped back in again.)
- IBM control blocks that are documented as having fixed real storage locations.

Examples:

FORMAT REAL(IEFBR14)
This command displays the real storage location of the IEFBR14 load module. Warning, it is likely that the real storage location of this module will change unpredictably and without warning. In fact, it is possible for its location to change even before the completion of the FORMAT command's processing. In other words, what is displayed is **not** guaranteed to still be correct by the time you see it!

FORMAT REAL(STCB.STCBLSDP?)-A0
FORMAT STCB.STCBLSDP?-A0
These commands display the current linkage stack entry (assuming that the current entry either is a branch state entry or a program-call state entry). The first command shows the entry in real storage. The second command shows the same entry in virtual storage.

FORMAT STCB.STCBDUCV?
FORMAT REAL(STCB.STCBDUCV?)

FORMAT STCB.STCBDUCR?~ASID(REAL)
All three of these commands display the Dispatchable Unit Control Table (DUCT).
- The first command displays the DUCT in virtual storage.
- The second command displays the same DUCT at its real storage location by converting the resolved virtual address to its corresponding real address.
- The third command uses a different method to display the DUCT at its real address. In this case, the System determined real address of the DUCT is loaded from the STCBDUCR field and then interpreted as a real address (via the ASID(REAL) function call).

FORMAT .PSAPCCAV?
FORMAT REAL(.PSAPCCAV?)
FORMAT .PSAPCCAR?~ASID(REAL)
The first of these commands displays the current CPU's PCCA in virtual storage. The remaining two commands display the PCCA in real storage. Assuming that all three of these commands are executed from the same CPU engine, they all display the same PCCA.

FORMAT REAL(MYDCB)
This displays in real storage the location of "MYDCB". Please be aware that MYDCB's real location can change unpredictably unless MYDCB is located in a page that has been page-fixed **and** in an address space that is nonswappable. (Page-fixed pages located within swappable address spaces can and do change real storage locations when the address space is swapped out and then swapped back in again.)

Help FUNctions SAve4

The SAVE4() function is used by c/XDC during address expression analysis to save Alet information. Its purpose is to assist the LIST VARIABLES command with dereferencing FAR pointers.

c/XDC uses SAVE4 in conjunction with the ALET(USE4) function. This function is not intended for direct use by customers.

Help FUNctions S1-thru-s8

The **S1(...)** thru **S8(...)** functions extract from 1 to 8 bytes of data from storage, from a register, or from a PSW. They then propagate the value's hi-order bit (its sign bit) leftwards until the result is 64 bits wide. They then make the result available for use in the computation of an address expression's offset term.

The **S1(...)** thru **S8(...)** functions can appear only within an offset term of an address expression. They may not appear between terms. (See **HELP ADDRESSING SYNTAX OFFSETTERMS** for more information.)

See **HELP FUNCTIONS CHART** for a table that summarizes the placement rules for all

built-in functions.

The **S1(...)** thru **S8(...)** functions treat the extracted value as a signed binary number having a value that can range from -2**(n*8-1) up to +2**(n*8-1)-1 (where "n" is the number of bytes extracted). Thus:

- **S1(...)** can extract a value ranging from -128 to +127.
- **S4(...)** can extract a value ranging from -2,147,483,648 up to +2,147,483,647 (2 gigabytes).
- **S8(...)** can extract a value ranging from -9,223,372,036,854,775,808 up to +9,223,372,036,854,775,807 (8 exabytes).

If you need to extracted **unsigned** values, use the **X1(...)** thru **X8(...)** functions.

Syntax:

```
S1( addr-exp            )
S2  register-reference
S3  PSW-reference
S4
S5
S6
S7
S8
```

Aliases:

- **H2(...)** is an alias of **S2(...).**
- **F4(...)** is an alias of **S4(...).**

Operands:

S1(thru **S8(**
The number (S1 thru S8) indicates how many bytes are to be extracted.

addr-exp
This may be any valid address expression resolving to any virtual address in any address space or data space. It may also resolve to any real storage address.

register-reference
This may be the name of any register. It also may include a starting offset **within** the register, just so long as the starting offset plus the extraction width does not exceed the width of the register.

The given offset must be a simple positive numeric. It may not be either symbolic or a complex computation.

Z/XDC has ways to refer to all register types, general (both wide and narrow), access, control, floating point and vector. For all the gory details, see **HELP**

COMMANDS LIST REGISTERS.

Examples:

S2(R2+1) - Extracts bytes 1 and 2 (i.e. the 2nd and 3rd bytes) from 4-byte wide, retry level general register R2. The sign bit is then propagated to 64 bits.

S1(ERW2+6) - Extracts the 7th byte from 8-byte wide, error level general register two. The sign bit is then propagated to 64 bits.

S2(AR2+1) - Extracts bytes 1 and 2 (i.e. the 2nd and 3rd bytes) from retry level access register AR2. The sign bit is then propagated to 64 bits.

S3(FR2+6) - Not valid. This attempts to extract the 7th, 8th, and 9th bytes from floating point register FR2; however, floating point registers are only 8 bytes wide.

S4(R15+1+1) - Not valid. **+1+1** is not a simple numeric.

PSW-reference

Z/XDC has four terms that refer to PSWs:

- The terms **PSW** and **EPSW** refer to the 8-byte wide, retry level and error level classic PSWs, respectively.

- The terms **PSWE** and **EPSWE** refer to the 16-byte wide, retry level and error level PSWs, respectively.

A PSW reference may be any of the above.

See **HELP ADDRESSING PARSERS ASM PSWS** for complete details about referring to PSWs.

A reference may also include a starting offset **within** the PSW, just so long as the starting offset plus the extraction width:

- Does not exceed 4 for [E]PSWs,
- Does not exceed 8 for [E]PSWEs.

The given offset must be a simple positive numeric. It may not be either symbolic or a complex computation.

Examples:

S3(PSW+5) - Extracts Bytes 5, 6 and 7 (i.e. the 6th, 7th and 8th bytes) from The retry level PSW.

S4(EPSW+5) - Not valid. This attempts to extract the 6th, 7th, 8th and 9th bytes from the error level PSW; however, classic PSWs are only 8 bytes wide.

These extraction functions **(S1** thru **S8)** can appear only within the offset terms of an address expression. They may not appear between terms. (For more information, see **HELP ADDRESSING SYNTAX OFFSETTERMS.)**

See **HELP FUNCTIONS CHART** for a table that summarizes the placement rules for all

built-in functions.

Example:

Assume:
- An equate named **MIDLIST** has been create to reference the middle of a list.
- **UPDOWN** is the name of a field within a control block of yours.
- You have issued:
  - A **DMAP** command to load a map of that control block,
  - And a **USING** command to assign that map to the location of your control block in storage.

The **FORMAT MIDLIST+S1(.UPDOWN)** will do the following:
- The 1-byte value stored in your control block's **UPDOWN** field is either added to or subtracted from (according to the value's sign bit) the location of the **MIDLIST** equate.

  The resulting location is displayed.

Help FUNctions Width

The WIDTH(...) built-in function trims or pads a numeric value (such as an address) to a specified bit width from 1 to 64 bits wide. The result is then padded with zeros on the left back out to 64 bits.

When a ZAP or VERIFY command's second operand is address data (i.e. an address expression, not string data), and that 2nd address expression includes one or more WIDTH(...) function calls, then the final width of the resolved address data is set to a standard byte width, based in the bit width specified in the expression's final WIDTH(...) function call. For detailed information, see HELP COMMANDS ZAP ADDRESS.

Syntax:

```
WIDTH( number )
```

number
This is a decimal number ranging from 1 to 64. It specifies how wide to make the address calculated so far. That value will be either truncated or zero-padded on the left as necessary to achieve the desired width.

The WIDTH(...) function can appear in an address expression anywhere following the base term, but not within an offset term. It must appear only between terms. (See HELP ADDRESSING SYNTAX for information about the structure of address expressions.)

See HELP FUNCTIONS CHART for a table that summarizes the placement rules for all built-in functions.

Help FUNctions XADDr

The XADDR(...) built-in function resolves a given address expression and returns the expression's resolved value as a simple numeric. This value can then be used:

- Arithmetically within a computation within an address expression's offset term.
- Logically in a conditional breakpoint as an 8-byte wide value to be tested.
- Logically in a VERIFY command as an 8-byte wide value to be tested.

Syntax:

XADDR(addressexpression)

Alias - XADR

addressexpression
This is an arbitrary address expression. It is resolved and its value is returned as a simple numeric that can be added to, multiplied, divided, tested against, etc.

The XADDR(...) built-in function can be used in the following contexts:

- In a conditional breakpoint as a value to be tested.
- In a VERIFY command as a value to be tested.
- Within the offset terms of an address expression. (It may not be used either between terms or as a base. See HELP ADDRESSING SYNTAX OFFSETTERMS for more information.)

See HELP FUNCTIONS CHART for a table that summarizes the placement rules for all built-in functions.

When used within an address expression, the XADDR(...) function can occur within any Z/XDC command or built-in function that accepts address expressions as operands.

Examples:

FORMAT 10?-XADDR(+0)&FFF
The XADDR(...) built-in function makes it possible to align an address expression to a page boundary. This example illustrates that: The FORMAT command displays the start of the page of storage that contains the start of the CVT. Example: If the CVT starts at FD26B0, then this expression displays location FD2000.

FORMAT +1000-XADDR(+0)&FFF
This command displays the start of the next page following the location most recently displayed (i.e. following the location currently pointed to by Z/XDC's "Current Display Pointer." See HELP ADDRESSING IMPLICIT for more information.)

TRACE BY (XADDR(NXINST()-XADDR(+0)&FFF),EQ,0000000005104000)
This is a conditional trace that allows execution to proceed until a branching instruction is reached that will jump to any location within the page whose virtual address is X'5104000'. Please note the following:

- "XADDR(NXINST()-XADDR(+0)&FFF)" contains an address expression, but as a whole,

it is **not** an address expression. It is a numeric value being used in the "location" portion of a conditional expression.

- The first appearance of "XADDR" is as a numeric function. Its operand is the address expression: "NXINST()-XADDR(+0)&FFF". The function converts this expression to a simple 8-byte numeric value that can be compared against by the rest of the conditional expression (",EQ,0000000005104000)").

- The second appearance of "XADDR" is as a built-in function. Its operand is the address expression: "+0". This expression is resolved to the same location as was returned by the "NXINST()" built-in function, and then it is converted to a pure numeric value that can be manipulated numerically.

- The "&FFF" portion of the address expression then isolates the lo-order 3 nibbles of the value returned by "XADDR(+0)". This then is subtracted from the location returned by the "NXINST()" function. This yields the starting address of the page containing the next following machine instruction to be executed.

- The XADDR(...) numeric function then converts this location to a numeric value which then is compared by the rest of the conditional expression to see if it is X'0000000005104000'. If so, then the conditional expression results in being TRUE, and so user program execution is then halted, and Z/XDC sends a display to the user.

TRACE BY (XADDR(NXINST()),AND,000000007FFFF000,EQ,00000000005104000)
This accomplishes the identical result as the prior example. In the prior example, the clearing of the three lo-order nibbles was done via the address expression portion ("NXINST()-XADDR(+0)&FFF") of the command. In this example the clearing is done via the mask portion ("000000007FFFF000") of the conditional expression.

Note, there is a problem with both of the above two examples: If execution reaches the page via fall thru from the preceding page, then execution doesn't stop until it reaches a branching instruction that will branch to elsewhere within that page. If no such instruction is reached, then execution does not stop.

TRAP 5104000
TRACE BY (XADDR(NXINST()),AND,000000007FFFF000,EQ,00000000005104000)
This sequence solves the problem noted above.

TRAP 5104528
TRACE BY (XADDR(NXINST()-05104528),LT,000000000000DF27)
This sequence traps program execution at any branch (or fall thru) that will cause execution to enter the range of storage between X'5104528' and X'511244E' (inclusive). (Note that "LT" is a **logical** compare, and so a negative value compares high against a positive value.)

Help FUNctions XADR

The XADR(...) function is an alias of XADDR(...). See HELP FUNCTIONS XADDR for more information.

Help FUNctions XALet

The XALET(...) built-in function returns an ALET as a simple numeric value. This value can then be used:

- Arithmetically within a computation within an address expression's offset term.
- Logically in a conditional breakpoint as a 4-byte wide value to be tested.
- Logically in a VERIFY command as a 4-byte wide value to be tested.

The XALET(...) function's operand may be either a register name or an address expression:

- If it is an address expression, then it is resolved to a location within an address space or a data space. If the resolved location was reached by use of an ALET, then this function returns that ALET as a simple numeric value. Otherwise, this function returns a zero value.

- Instead of an address expression, a pure register name may be given directing the function to presume that the register's contents is an ALET and to return the contents as a simple numeric value. (The function makes no effort to determine whether or not the returned value actually is a legal ALET.)

Syntax:

```
XALET(addressexpression)
     (register)
```

addressexpression
This is an arbitrary address expression. It is resolved, and then its associated ALET (if any) is returned as a simple numeric value that can added to, multiplied, divided, etc.

If the selection of the address expression's final target data/address space required the resolution of an ALET, then that ALET is the ALET that is returned by this XALET(...) function. Otherwise, no ALET is associated with the address expression, and so the XALET(...) returns a numeric zero.

register
If a pure register name is given, then its contents are presumed to be an ALET. So that ALET is returned (without inspection) as the function's result.

The only permitted register types are 4-byte wide registers:
- Access registers (ARn EARn)
- The low halves of general registers (Rn ERn)
- The high halves of general registers (RHn ERHn)

The XALET(...) built-in function can be used in the following contexts:

- In a conditional breakpoint as a value to be tested.
- In a VERIFY command as a value to be tested.
- Within the offset terms of an address expression. (It may not be used either

between terms or as a base. See HELP ADDRESSING SYNTAX OFFSETTERMS for more information.)

See HELP FUNCTIONS CHART for a table that summarizes the placement rules for all built-in functions.

When used within an address expression, the XALET(...) function can occur within any Z/XDC command or built-in function that accepts address expressions as operands.

Examples:

DMAP XDCMAPS.DUCT
USING DUCT ECR2?ASID(REAL) FLOAT
EQUATE DUAL ASID(REAL)0+X4(DUCT+10)&7FFFFF80 FLOAT HEX
EQUATE ALE_AR4 DUAL+XALET(AR4)&FFF*10 FLOAT HEX 10
FORMAT ALE_AR4
This sequence of commands locates and displays the Access List entry (located on the current Dispatchable Unit Access List) "pointed to" by the Access List Entry Token (ALET) located in access register AR4. Note, this assumes (a) AR4 actually does contain an ALET and (b) that ALET references the Dispatchable Unit's Access List and NOT the address space's Primary Access List. These commands work as follows:

DMAP XDCMAPS.DUCT
Reads a dsect map (from a load module named XDCMAPS) of the Dispatchable Unit Control Table.

USING DUCT ECR2?ASID(REAL) FLOAT
Assigns the DUCT dsect to represent the real storage pointed to by hardware control register CR2.

EQUATE DUAL ASID(REAL)0+X4(DUCT+10)&7FFFFF80 FLOAT HEX
Defines a label (DUAL) to represent, in real storage, the location of the start of the Dispatchable Unit Access List.

EQUATE ALE_AR4 DUAL+XALET(AR4)&FFFF*10 FLOAT HEX 10
Defines a label (ALE_AR4) to represent, in the DUAL, the location of the Access List Entry "pointed to" by the ALET contained within access register AR4. Note, only part of the ALET indexes into the Access List, so the ALET has to be extracted numerically ("XALET(AR4)") so that it can be manipulated appropriately.

Note, ALETs, ALEs, DUALs, and DUCTs are all documented in chapter 5 of IBM's **z/Architecture Principles of Operation** (SA22-7832).

TRAP .TABLECHK (XALET(AR5?),AND,FF000000,EQ,01000000)
TRAP .TABLECHK (XALET(AR5),AND,FF000000,EQ,01000000)
TRAP .TABLECHK (XALET(AR5),EQ,01)
These three commands produce the identical result. In all of these commands, the XALET function returns the ALET contained within access register AR5:
- In the first command, "AR5?" is an address expression that refers to an address (taken from **general** register R5) in an address/data space designated by the ALET in access register AR5. The XALET function then extracts and returns that address expression's ALET component (discarding the address component).
- In the second command, "AR5" is a pure register name. The XALET function presumes that the register contains a legal ALET, and so it returns that ALET as its

result.
- In the third command, the width of the constant limits the comparison to only the first byte of the ALET; the remaining three bytes are not tested.

In all of these cases, the same ALET is returned. The only difference between the first and second of these commands is that the first command will fail if the ALET is not legal, while the second command will not.

In any case, these commands set a conditional trap at the location labeled TABLECHK. The trap halts user program execution when the ALET contained within access register AR5 references any entry located in the address space's **primary** access list (not the task's access list).

Help FUNctions XASId

The XASID(...) built-in function returns an ASID as a simple numeric value. This value can then be used:

- Arithmetically within a computation within an address expression's offset term.
- Logically in a conditional breakpoint as a 2-byte wide value to be tested.
- Logically in a VERIFY command as a 2-byte wide value to be tested.

The XASID(...) function's input parameter is an address space reference (referred to by Z/XDC as **aspaceref).** that can be an ASID number, a jobname, a keyword or any of several other things. The referenced space may be any address space (security is not relevant), or it may be real storage. (It may not be a data space.) The complete syntax for aspaceref parameters can be found at HELP COMMANDS SYNTAX ASIDS.

Syntax:

XASID(aspaceref)

Alias - XASN

aspaceref
This parameter identifies the address space which the current address expression is to reference from this point onwards. Briefly, aspaceref can be any of the following:
- A jobname.
- A keyword indicating the home space or the retry level or error level, primary, secondary, or instruction execution address space. Examples: HOME, PASID, ESASID, IASID, etc.
- The keyword REAL, indicating real storage.
- An ASID number.
- An address expression pointing to an ASID number.
- A register containing an ASID number.
- The name of an equate or dsect that has been assigned to represent storage in an address space. (The assigned address space is used.)
- A null value indicating that the home address space is to be referenced.

For detailed information, see HELP COMMANDS SYNTAX ASIDS.

The XASID(...) built-in function can be used in the following contexts:

- In a conditional breakpoint as a value to be tested.
- In a VERIFY command as a value to be tested.
- Within the offset terms of an address expression. (It may not be used either between terms or as a base. See HELP ADDRESSING SYNTAX OFFSETTERMS for more information.)

See HELP FUNCTIONS CHART for a table that summarizes the placement rules for all built-in functions.

When used within an address expression, the XASID(...) function can occur within any Z/XDC command or built-in function that accepts address expressions as operands.

Examples:

```
DMAP XDCMAPS.CVTFIX
DMAP .ASVT
DMAP .ASCB
USING .CVTTCBP 10?
USING ASVT .CVTASVT?
USING ASCB .ASVTFRST+XASID(EPASID)*4? FLOAT
FORMAT ASCB
```

This sequence of commands locates and displays the ASCB for the error level instance of the Primary Address Space. It works as follows:

DMAP XDCMAPS.CVTFIX
DMAP .ASVT
DMAP .ASCB
Reads dsect maps (from a load module named XDCMAPS) of the CVT, the ASVT, and an ASCB.

USING .CVTTCBP 10?
USING ASVT .CVTASVT?
Assigns the CVT and ASVT dsects to map the System’s Communications Vector Table and Address Space Vector Table, respectively.

USING ASCB .ASVTFRST+XASID(EPASID)*4? FLOAT
The XASID(EPASID) function returns the ASID of the error level instance of the Primary Address Space. The rest of the address expression uses that ASID to index the ASVT to find the pointer to the corresponding Address Space Control Block and assign the ASCB dsect to map that control block.

FORMAT ASCB
This command produces a formatted display of that ASCB.

```
TRACE BY (XASID(NXINST(PC)),EQ,0002)
```

This trace allows user program execution to proceed until a machine instruction is reached that will cause execution to transfer to the address space whose ASID=0002. (See HELP FUNCTIONS NXINST for more information about the NXINST function.)

```
TRACE BY (XADDR(0+XASID(NXINST(PC))-XASID(PCAUTH)),EQ,00000000)
```

This trace allows user program execution to proceed until a machine instruction is

reached that will cause execution to transfer to the PCAUTH address space. This example uses the XADDR numeric function against an address expression that uses the XASID built-in functions. More specifically, this example works as follows:

- **NXINST(PC)**
  This is an address expression that resolves to a location (address **and address space)** of the machine instruction that will be executed next after the instruction pointed to by the then current retry level PSW. In other words, at each branching instruction that **will** in fact branch, and at each PC, SVC, TRAP2, and TRAP4 instruction, the NXINST(PC) function returns the target address and address space of the branch or transfer that is about to occur. (See HELP BREAKPOINTS TRACING BRANCHES for more information about "TRACE BY" type traces.)

- **XASID(NXINST(PC))**
  This extracts the ASID component of the "NXINST(PC)" address expression and returns it as a numeric value for use as an offset term of the larger address expression (i.e. "xxx" of "0+xxx-yyy". See HELP ADDRESSING SYNTAX OFFSETTERMS.)

- **XASID(PCAUTH)**
  This returns the ASID of the PCAUTH address space. It is returned as a numeric value for use as an offset term of the larger address expression (i.e. "yyy" of "0+xxx-yyy").

- **XASID(NXINST(PC))-XASID(PCAUTH)**
  This subtraction (of two offset terms) essentially compares the two extracted ASIDs for equality.

- **0+XASID(NXINST(PC))-XASID(PCAUTH)**
  This is a syntactically valid address expression that resolves to a location whose address equals the difference between the two extracted ASIDs. If the two ASIDs are equal, then this expression will resolve to location zero. **However,** since it technically is a **location** and **not** a numeric value, it cannot yet be used as a comparison operand in the conditional expression. So ...

- **XADDR(0+XASID(NXINST(PC))-XASID(PCAUTH))**
  This uses the XADDR numeric function to extract the address expression's address component as a 4-byte wide numeric value. This (finally) is a value that can be compared against by the conditional expression.

Help FUNctions XASN

The XASN(...) function is an alias of XASID(...). See HELP FUNCTIONS XASID for more information.

Help FUNctions X1-thru-x8

The **X1(...)** thru **X8(...)** functions extract from 1 to 8 bytes of data from storage, from a register or from a PSW. They then pad the extracted value with zeros on the left out to 64 bits and then make the result available for use in the computation of an address expression's offset term.

The **X1(...)** thru **X8(...)** functions can appear only within an offset term of an address expression. They may not appear between terms. (See **HELP ADDRESSING SYNTAX OFFSETTERMS** for more information.)

See **HELP FUNCTIONS CHART** for a table that summarizes the placement rules for all built-in functions.

The **X1(...)** thru **X8(...)** functions treat the extracted value as an unsigned binary number having a value that can range from zero up to 2**(n*8)-1 (where "n" is the number of bytes extracted). Thus:

- **X1(...)** can extract a value ranging from 0 to 255.
- **X4(...)** can extract a value ranging from 0 to 4,294,967,295 (4 gigabytes).
- **X8(...)** can extract a value ranging from 0 to 18,446,744,073,709,551,615 (16 exabytes).

If you need to extracted **signed** values, use the **S1(...)** thru **S8(...)** functions.

Syntax:

```
X1( addr-exp           )
X2  register-reference
X3  PSW-reference
X4
X5
X6
X7
X8
```

Operands:

X1(thru **X8(**
The number (X1 thru X8) indicates how many bytes are to be extracted.

addr-exp
This may be any valid address expression resolving to any virtual address in any address space or data space. It may also resolve to any real storage address.

register-reference
This may be the name of any register. It also may include a starting offset **within** the register, just so long as the starting offset plus the extraction width does not exceed the width of the register.

The given offset must be a simple positive numeric. It may not be either symbolic or a complex computation.

Z/XDC has ways to refer to all register types, general (both wide and narrow), access, control, floating point and vector. For all the gory details, see **HELP COMMANDS LIST REGISTERS.**

Examples:

X2(R2+1) - Extracts bytes 1 and 2 (i.e. the 2nd and 3rd bytes) from 4-byte wide, retry level general register R2. The extraction is then zero padded on the left out to 64 bits.

X1(ERW2+6) - Extracts the 7th byte from 8-byte wide, error level general register two. The extraction is then zero padded on the left out to 64 bits.

X2(AR2+1) - Extracts bytes 1 and 2 (i.e. the 2nd and 3rd bytes) from retry level access register AR2. The extraction is then zero padded on the left out to 64 bits.

X3(FR2+6) - Not valid. This attempts to extract the 7th, 8th, and 9th bytes from floating point register FR2; however, floating point registers are only 8 bytes wide.

X4(R15+1+1) - Not valid. **+1+1** is not a simple numeric.

PSW-reference

Z/XDC has four terms that refer to PSWs:

- The terms **PSW** and **EPSW** refer to the 8-byte wide, retry level and error level classic PSWs, respectively.

- The terms **PSWE** and **EPSWE** refer to the 16-byte wide, retry level and error level PSWs.

A PSW reference may be any of the above.

See **HELP ADDRESSING PARSERS ASM PSWS** for complete details about referring to PSWs.

A reference may also include a starting offset **within** the PSW, just so long as the starting offset plus the extraction width:

- Does not exceed 4 for [E]PSWs,
- Does not exceed 8 for [E]PSWEs.

The given offset must be a simple positive numeric. It may not be either symbolic or a complex computation.

Examples:

X3(PSW+5) - Extracts bytes 5, 6 and 7 (i.e. the 6th, 7th and 8th bytes) from the retry level PSW.

X4(EPSW+5) - Not valid. This attempts to extract the 6th, 7th, 8th and 9th bytes from the error level PSW; however, PSWs are only 8 bytes wide.

These extraction functions **(X1** thru **X8)** can appear only within the offset terms of an address expression. They may not appear between terms. (For more information, see **HELP ADDRESSING SYNTAX OFFSETTERMS.)**

See **HELP FUNCTIONS CHART** for a table that summarizes the placement rules for all built-in functions.

Examples:

EQUATE DDENT TIOT+X2(IHADCB.DCBTIOT) D
FORMAT DDENT
Assuming that "TIOT" labels the start of a Task I/O Table, and assuming that "IHADCB" is a dsect map that maps an opened Data Control Block, then the EQUATE command assigns a label named "DDENT" to the TIOT's DD entry that corresponds to the opened DCB. The FORMAT command then displays that DD entry.

USING ASCB ASVT.ASVTFRST+X2(HISASID)*4? F
Assume that an ASID number has been stored at a location labeled "HISASID". Then this command assigns an ASCB dsect map to represent the location of the ASCB associated with the stored ASID.

FORMAT .KEYTABLE+X1(PSW+1)/10*0C
Assume that the following table has been assembled into the user program:

```
    KEYTABLE DC CL12'SYSTEM'
             DC CL12'JES2'
             DC CL12'VSPC'
             DC CL12'RESERVED'
             DC CL12'RESERVED'
             DC CL12'DATAMGMT'
             DC CL12'VTAM/TCAM'
             DC CL12'IMS'
             DC CL12'USER KEY8'
             DC CL12'USER KEY9'
             .
             .
             .
```

Then this command displays the table entry that corresponds to the retry level PSW's current execution key. The address expression is processed as follows:

- **X1(PSW+1)** extracts from the retry level PSW the byte that contains the execution key. The key value is in the hi-order 4 bits of the byte.
- **/10** shifts the key value rightwards by 4 bits to position it in the lo-order 4 bits of Z/XDC's work area. It also purifies the key value by discarding the various flag bits that had been in the shifted out bit positions.
- ***0C** converts the key value to a table index for the KEYTABLE table.
- The index is then added to the address of KEYTABLE to yield the location of the table slot corresponding to the execution key.

Help FUNctions ZLeft

The ZLEFT(...) built-in function zeros out a specified number of a value's left-most bits. The value can be either a numeric value or an address. For the purposes of this function, all values are considered to be 64 bits wide.

Syntax:

```
ZLEFT( number )
```

number
This is a decimal number ranging from 1 to 64. It specifies how many left-hand bits of the address calculated so far are to be zeroed.

The ZLEFT(...) function can appear in an address expression anywhere following the base term, but not within an offset term. It must appear only between terms. (See HELP ADDRESSING SYNTAX for information about the structure of address expressions.)

See HELP FUNCTIONS CHART for a table that summarizes the placement rules for all built-in functions.

Example:

FORMAT R1?~ZLEFT(48)
R1 refers to only the lo-order 32 bits of general register R1; however, the address expression **R1?** (like all address expressions in Z/XDC) is 64 bits wide.
The **~ZLEFT(48)** function zeros out the left-most 48 bits of the address expression, leaving only the lo-order 16 bits intact. So, for example, if R1 had contained X'87009A3E', then location 00000000_00009A3E would be be displayed.

Help FUNctions ZRight

The ZRIGHT(...) built-in function zeros out a specified number of a value's right-most bits. The value can be either a numeric value or an address.

Syntax:

```
ZRIGHT( number )
```

number
This is a decimal number ranging from 1 to 64. It specifies how many right-hand bits of the address calculated so far are to be zeroed.

The ZRIGHT(...) function can appear in an address expression anywhere following the base term, but not within an offset term. It must appear only between terms. (See HELP ADDRESSING SYNTAX for information about the structure of address expressions.)

See HELP FUNCTIONS CHART for a table that summarizes the placement rules for all built-in functions.

Example:

FORMAT R1?~ZRIGHT(16)
The right-most 16 bits of the address expression are zeroed out, leaving only the hi-order 48 bits intact. (All address expressions in Z/XDC are 64 bits wide.) So, for example, if R1 had contained X'874F9A3E', then **R1?** would yield an address of

00000000_874F9A3E, and zeroing out the right-most 16 bits would yield 00000000_874F0000. That then is the location that would be displayed.

Help ATtentions

When Z/XDC is being use to debug a program, it can receive two distinct classes of attention interrupts: those that occur while Z/XDC is in control and those that occur while the user program is in control. Attentions received while Z/XDC is in control can be used to abort certain potentially long running processes such as the reading of symbol maps (the **MAP** and **DMAP** commands) or the scanning of storage (the **FIND** command).

Z/XDC can always receive attentions that are signaled while it is in control; however, it cannot always receive attentions when the user program is in control. Whether or not Z/XDC can receive such attentions depends on the overall environment in which Z/XDC is being used:

- In TSO, attention support is provided by the **XDCCALL** command. When the debugging session is started using **XDCCALL** or its aliases or Z/XDC's Startup Panel in ISPF (which uses the **XDCCALL** command), Z/XDC can receive attention interrupts issued while the user program is in control. In other words, the user can use the attention key to interrupt the user program and pass control to Z/XDC.

- In the batch, attention support is provided by cdf/XDC. If cdf/XDC is being used for interactive communication with the user program, then the attention key can be used either to disconnect the user's terminal from the debugging session (without interrupting the user program) or to interrupt the user program and pass control to Z/XDC. (For more information, see **HELP XDCSRVER CDF CONNECTION.)**

Attentions **cannot** be used to interrupt a user program under the following circumstances:

- In TSO, if neither the **Z/XDC Startup Panel** in ISPF nor the **XDCCALL** command (or its aliases) was used to start the debugging session.

- In the batch, if interactive communication with Z/XDC is being performed by operator console (WTOs and WTORs) instead of with cdf/XDC.

When these restrictions apply, they only apply when the user program is in control of execution. As mention above, Z/XDC can always receive attentions that are signaled while it, instead of the user program, is in control of execution.

Warning: When an attention interrupt is received by Z/XDC during user program execution, the environment in which Z/XDC receives control is **not** that of the user program. Instead, it is of the attention handling routine that was triggered by the attention signal. Consequently, Z/XDC's commands that show or use registers or the PSW will use the registers and PSW belonging to the attention handler routine, not those belonging to the user program. See **HELP ATTENTIONS INUSERPGM** for more information.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

INXDC - Using attentions to interrupt long running Z/XDC commands.
INUSERPGM - Using attentions to interrupt a user program and pass control to Z/XDC.
INTHEBATCH - Using attentions when connected via cdf/XDC to a debugging session running in a batch job or started task.

Help ATtentions INXdc

Generally, Z/XDC ignores attentions received while it is in control; however, during certain long running commands, Z/XDC will check for attention events and will cleanly abort the command if it sees that an attention has been signaled.

Z/XDC will check for attentions occurring during the processing of the MAP, DMAP, and FIND commands, and during the processing of the ILC and NXSEQ built-in functions.

When Z/XDC is running in TSO, or when Z/XDC is running in a background job and communicating with the user via cdf/XDC, attentions can be signaled in the manner that is normal for TSO and the current terminal.

However, when Z/XDC is running in a background address space (batch or system task), and it is using operator consoles and WTO/WTORs for communicating with the user, the System does not provide attention mechanisms from operator consoles. Accordingly, in this situation Z/XDC issues an extra WTOR: DBC839I REPLY NULL FOR ATTENTION INTERRUPT. (This appears in addition to the prompting WTOR that Z/XDC uses to received commands.) If Z/XDC is processing a long running command or function, and it is taking too long to complete, then the user can type any character in response to the "attention WTOR", and this will cause Z/XDC to abort the current command and await a new command.

Help ATtentions INUserpgm

Z/XDC cannot always receive attentions when the user program is in control. Whether or not Z/XDC can receive such attentions depends on the overall environment in which Z/XDC is being used:

- In TSO, attention support is provided by the **XDCCALL** command. When the debugging session is started using **XDCCALL** or its aliases or Z/XDC's Startup Panel in ISPF (which uses the **XDCCALL** command), Z/XDC can receive attention interrupts issued while the user program is in control. In other words, the user can use the attention key to interrupt the user program and pass control to Z/XDC.

- In the batch, attention support is provided by cdf/XDC. If cdf/XDC is being used for interactive communication with the user program, then the attention key can be used either to disconnect the user's terminal from the debugging session (without interrupting the user program) or to interrupt the user program and

pass control to Z/XDC. For more information, see HELP XDCSRVER CDF CONNECTION.

For information on when attentions **cannot** be used to interrupt a user program, type HELP ∗UP.

The Attention Processing Environment

Warning: When an attention interrupt is received by Z/XDC during user program execution, the environment in which Z/XDC receives control is **not** that of the user program. Instead, it is of the attention handling routine that was triggered by the attention signal. Consequently, Z/XDC's commands that show or use registers or the PSW will use the registers and PSW belonging to the attention handler routine, not those belonging to the user program. Specifically, this means the following:

- The registers and PSW (as shown by **LIST REGS, LIST PSW, LIST EREGS, LIST EPSW** and similar commands) belong to the attention handling routine; they do not belong to the user program.

  Use of **Rn** (e.g. **R13), ERn, PSW, EPSW** etc. will refer to an attention routine register or PSW, not to a user program register or PSW.

- The **retry level,** as shown by the **LIST RBS** command, will be Z/XDC's Interruption Handling Routine. The interrupted user program will actually be one of the following:

  - If you are debugging a program running in TSO, the interrupted program will be that which is controlled by the Request Block that is next older to the retry level IRB. Use **LIST RBS** to display the Request Blocks queue.

  - If you are debugging a program running in a background job and are connected to it via cdf/XDC, then the program you wish to interrupt will be running in the parent task to the task in which Z/XDC is running:
    - Use **LIST TASKS** to see what that parent task is.
    - Then use **LIST RBS TCB#n** to display that task's Request Block queue. The newest displayed task will most likely be the one that you will want to interrupt.

- While in the Attention Handling Routine, attempts to simply trace execution will actually trace the rest of the Handling routine. And I promise you, that will go badly!

Whenever an attention is signaled while the user program is in control, Z/XDC's Interruption Handling Routine will receive control, and it will display message **DBC893I** along with about **six paragraphs(!)** explaining what the current environment is and how you can proceed back to a normal environment. The information provided is both detailed and **customized** for the specific interruption environment. If you read the information and follow the instructions carefully, you should be able to find your way back to the interrupted routine.

The **DBC893I** messages explain exactly how to do all that. See **HELP MESSAGES DBC893** for more information.

User Program WAIT States

If, when you pressed the **ATTN** key, your program was in a WAIT state, Z/XDC's Interruption Handler will run and process as described above; however, unless further action is taken, your program will remain in that WAIT state undisturbed, even after you use the **GO** command to allow the Interruption Handler to complete and go away.

In this case, you might **not** want to resume Z/XDC in its normal environment (as described above), but instead just issue Z/XDC commands to look around from within the Handler's environment. If this is what you want to do, then you must understand the following:

- When Z/XDC gains control via the **ATTN** key, Z/XDC is running in a **different** environment than the program that was interrupted. That program is suspended, but unless you zap storage or set suitable breakpoints, that program will continue its WAIT state (or resume its execution) as soon as the attention routine completes.

- You may display any user program code and data areas you like. What you will see will be current and correct. However, if you wish to display the user program **registers and PSW,** you must first find the Request Block under which your program is running:

- In TSO, that will be the RB that is next older to the IRB under which Z/XDC's Interruption Handler is running. Use **LIST RBS** to display the current task's Request Blocks queue.

- In background jobs, that will be the newest RB running under the current task's **parent** task:
  - Use **LIST TASKS** to find the parent task.
  - Then use **LIST RBS TCB#n** to display that task's Request Blocks queue.

  Once you have found the correct Request Block:
  - Use **LIST REGS RB#n** to display that routine's registers.
  - Similarly, use **LIST PSW RB#n** to display that routine's PSW.

  Note, **RB#n** is a series of equates that the **LIST RBS** command automatically creates to label the locations of the Request Blocks being reported. See **HELP EQUATES BUILTIN AUTOMATIC** for more information.

On the other hand, you may instead want to force your program out of its WAIT state. If this is the case, then there two ways you might try going about doing that:

- After looking around, you might be able to discover whatever it is that your program is WAIT'ing on. With that knowledge, you might then be able to manually trigger that event. You might also use the **TRAP** command to set breakpoints at points of interest (such as your program's resume point after a WAIT SVC). Then you should use the **GOT** command to cause the attention exit to cleanup and go away, thereby allowing the user program to receive the POST and resume running. If this is possible, then this is the best thing to do.

- Alternatively, if you have no idea what might be needed to resolve the WAIT state, then you might consider using Z/XDC's **POST** command to force the WAIT state to complete. However, doing this will only make your program **think** that whatever it was waiting upon has occurred. The **POST** command has no way of

actually performing the missing action. So use the **POST** command only with caution.

Help ATtentions INThebatch

When you are connected (via cdf/XDC) to a debugging session that is running in a batch job or in a started task, your terminal's keyboard will be unlocked only while Z/XDC is in control.

If **your** program is in control (for example if you have issued a **GO** command to let your program resume execution), then your keyboard will be locked (because it is Z/XDC that owns the terminal connection, not your program).

However, if you wish, you can use the attention key **(ESC** on most keyboards these days) to unlock the keyboard and receive a panel giving you options for what you want cdf/XDC to do with your attention signal.

For further details, please see **HELP XDCSRVER CDF CONNECTION.**

Help Breakpoints

A **breakpoint** is a Z/XDC device for deliberately interrupting the execution of a user's program and causing control to pass to Z/XDC. Breakpoints can be placed essentially anywhere in a user program's execution path.

When a breakpoint is reached by user program execution, a hardware interrupt occurs that causes the program's status (PSW, registers, states and modes) to be saved and eventually causes Z/XDC to receive control, whereupon Z/XDC sends a display to a terminal and awaits commands from a user.

Two Types of Breakpoints: s0C1 ABENDs vs. TRAP2 Instructions

Z/XDC creates a breakpoint in a user's program by changing a target machine instruction's first one or two bytes...
- Either to **X'00'** (an illegal opcode),
- Or to **X'01FF'** (a TRAP2 instruction).

So when user program execution eventually reaches that instruction...

- Either a **0001 program check interrupt** occurs which will eventually cause Z/XDC to receive control from the System's Recovery/Termination Manager (RTM) with an indication that an s0C1 ABEND has occurred.

- Or a **Trap Exception** occurs which will cause the hardware immediately to pass control to a Z/XDC-owned Trap Handler Routine which will very quickly pass control to Z/XDC proper.

The trapping method used is controlled by the **TRAP2** and **ZERO** operands of the **SET TRACE** command. See HELP COMMANDS SET TRACE TRAP2 for more information.

Breakpoints via s0C1 ABENDs

In order for the s0C1 method to work, Z/XDC must be set up as your program's newest ABEND Recovery routine (ARR, ESTAE, ESTAEX, ESTAI, etc.). Then when the 0001 program check occurs, the program is not terminated. Instead, the resulting s0C1 ABEND is intercepted, and the System passes control to Z/XDC as an ESTAE-type (or FRR-type) ABEND Recovery routine.

Breakpoints via TRAP2 Instructions

On the other hand, if the TRAP2 method is being used, then the Z/XDC-owned Trap Handler Routine receives control directly from the hardware, and so RTM and other System overhead is avoided.

In this case, Z/XDC is running as a Trap Handler Routine (not as an ABEND Recovery Routine).

At first blush, it would appear that making Z/XDC your program's newest ABEND Recovery Routine would not be necessary when TRAP2-type breakpoints are being used, but that is not actually the case. This is because in order for TRAP2 instructions to work, a Trap Handler Routine must be installed for each execution thread (Task or SRB) to be debugged.

When such a Handler is not installed, the TRAP2 instruction does not work. Instead, a Special Operation Exception occurs (program interrupt code 0013), and that eventually leads to an s0D3 ABEND.

But if Z/XDC is set up as your newest ABEND Recovery Routine, it will see that s0D3 ABEND, and it will be able to recover from it! It does this by dynamically installing the missing Trap Handler Routine and then resuming execution to reattempt the TRAP2.

Once the Trap Handler Routine is in place, Z/XDC no longer needs to be your program's newest Recover routine... at least not for the purposes of trapping and tracing your code. But...

- It will still be needed to capture any other ABENDs that your program may suffer.
- And for multi-threaded programs, it will still be needed for setting up Trap Handler Routines in any other of your tasks (and SRBs) that you may wish to debug.

Error Level and Retry Level Execution Environments Considerations

When an ABEND occurs, The System's Recovery/Termination Manager (RTM) gains control and passes control to an ABEND Recovery Routine (such as Z/XDC), and he provides to the Recovery Routine (Z/XDC) information about two distinct environments:

- The **Error Level** environment consists of the registers, PSW and state data (AMODE, ASC-MODE, execution state, Request Block queue depth, etc.) **that existed** at the moment that the program ABENDed.

- The **Retry Level** environment consists of the registers, PSW, state data and Request Block level that **will exist** in the event that the Recovery Routine tells RTM to allow the ABENDed program to resume execution.

There are many very good reasons why these two environments are not the same. For details, please read HELP EXECUTIONLEVELS.

But remember! The distinction between the Error and Retry Levels is an **artifact** of the ABEND and recovery process. So it arises for 0C1-type breakpoints. It does **not** arise for **TRAP2-type** breakpoints!

Error vs. Retry Level Considerations Arising from 0C1-Type Breakpoints

When Z/XDC is in control due to an **0C1-type** breakpoint, The Error Level and retry Level environments may be the same, or they may be different. When they are **different,** the usefulness of Z/XDC is limited:

- When you use the **LIST RBS** command, The Error Level Request Block will be separately captioned.

- You have to use the **EWHERE EPSW?-2** command to view the Error Level execution location.

- You can use the **LIST ERWREGS** command (for example) to view the Error Level general registers.

- You will **NOT** be able to use the **TRACE** command to step execution through Error Level code. Only Retry Level code can be stepped through.

- You will **NOT** be able to use the **GO** command to resume Error Level execution. Retry Level execution will be resumed (not Error Level execution).

On the other hand, when the Error Level and Retry Level environments are **the same,** everything is much better:

- When you use the **LIST RBS** command, The Error Level Request Block will not be separately captioned. Only the Retry Level Request Block will be labeled.

- You can use the **WHERE** command to view the current execution location.

- You can use the **LIST RWREGS** command (for example) to view the Retry Level general registers.

- You **will** be able to use the **TRACE** command to step execution through your code.

- You **will** be able to use the **GO** command to resume your program’s execution.

Error vs. Retry Level Considerations Go Away for TRAP2-Type Breakpoints

When Z/XDC is in control due to a **TRAP2-type** breakpoint, there is no distinction between the Error Level and Retry Level environments...

- All of Z/XDC's Error Level constructs (EWHERE command, ERn register names, etc.) will still work, but they will display generally the same information as the Retry Level constructs.

- A **LIST RBS** command will not show any **(ARTIFACT)** RBs:
  - There will be no **PGM CHK** RB for running the RTM (IGC0101C),
  - There will be no **SYNCH** RB for running Z/XDC.
  - The only Request Blocks that will be shown are those that actually belong to the program being debugged (and whatever System Services it may have invoked).

- All Retry Level data (registers, PSWs and states) will initially be **identical** to the Error Level data (except the Retry Level PSW will have been adjusted backwards by two bytes).

- You will be able to use the **TRACE** command freely to step through your program's execution.

- You will be able to use the **GO** command freely to resume your program's execution.

What Can You Debug?

Because Z/XDC is essentially nothing more than a glorified ESTAE or FRR routine (or a TRAP handler), it can be used to debug any program code that runs in an environment where an ESTAE can be established (generally task mode, non-SRB mode code). It also can be used in many environments where an FRR can be established - SRB routines, locked code, FRR protected tasks and the like.

Integrating Z/XDC into Programs with Preexisting Recovery Routines

However, in order for Z/XDC to be useful, it generally must be the **newest** recovery routine (ESTAE-type or FRR) in existence at the moment that:
- An s0C1-type breakpoint is reached,
- A TRAP2-type breakpoint is reached for the first time within a task or SRB routine,
- Any other (non-breakpoint related) ABEND is suffered by your program.

If Z/XDC is **not** the newest recovery routine (e.g. if another ESTAE macro has been issued more recently than the ESTAE macro that was issued to setup Z/XDC), then when an ABEND occurs (or when a breakpoint is reached), the newer ESTAE will receive control ahead of Z/XDC and will, in all probability, handle the ABEND in ways that are not appropriate to the needs of a debugging session. In other words, hilarity will ensue.

Consequently, in order to use Z/XDC's breakpoints successfully, it is important that Z/XDC be properly integrated into that portion of your program that provides ABEND Recovery support. In other words:

- If the program that you want to debug issues ESTAE or ESTAEX macros,

- Or if it issues ATTACHs with ESTAI= operands,

- Or if it makes use of Request Block driven exit routines (such as attention exits, timer exits, DCB exits, etc.),

Then in order to use Z/XDC fully, you will either have to modify your code to make Z/XDC the newest recovery routine or (and this is a far easier thing to do) set a **hook** into your code at the point that you want to start debugging.

Note that even if you are using TRAP2-type breakpoints, you still need to make Z/XDC the newest recovery routine so that it will be able to automatically install its Trap Handler Routine when needed.

The Old way: Modify Your Code

Modify your program so that whenever it ABENDs or reaches a breakpoint, Z/XDC will be the first ABEND Recovery routine in line to processing the resulting interrupt. Possible modifications may include various of the following. (Follow the crosslinks for more detailed information):

- When running your program within an XDCCALL created environment, refrain from establishing any ABEND Recovery routines at all. **Or...**

- Whenever you issue an ESTAE or ESTAEX macro for your own recovery routine, follow it with a 2nd ESTAE or ESTAEX macro identifying Z/XDC as an even newer recovery routine. Or...

- Add code to the start of your own recovery routine that conditionally BASRs to Z/XDC.

- In order to debug certain exit routines (DCB OPEN exits, timer exits, attention exits, etc.), add an ESTAE or ESTAEX macro near the start of the exit's code to establish Z/XDC as that exit's recovery routine. (This is because these exits run under the control of their own Request Blocks [RBs] different from the RB under which your main program is running.)

- When ATTACH'ing a subtask, use the ESTAI= operand to setup Z/XDC as the subtask's initial recovery routine. (This may not be necessary if your program is running in an XDCCALL created environment.)

IN ALL OF THE ABOVE LISTED SITUATIONS, IF A BREAKPOINT IS SET WITHOUT FIRST HAVING TAKEN THE STEPS NECESSARY TO MAKE Z/XDC THE LOCAL ENVIRONMENT'S NEWEST RECOVERY ROUTINE, THEN WHEN THE BREAKPOINT IS REACHED BY USER PROGRAM EXECUTION, THE RESULTING PROGRAM INTERRUPT WILL NOT BE PROPERLY HANDLED BY EITHER THE OPERATING SYSTEM OR BY USER WRITTEN RECOVERY ROUTINES. THE RESULTS WILL RANGE FROM CONFUSING TO UTTERLY MYSTERIOUS!

Note that even if you are using TRAP2 instructions as breakpoints, you still need to set up Z/XDC as your newest recover routine so that it can automatically install the Trap Handling Routine when necessary.

The New Way: Use Hooks

Z/XDC's **HOOK** command in most cases offers a **much easier alternative** for setting up a local Z/XDC environment. If the local environment into which you want to place a breakpoint does not have Z/XDC properly established as its newest ABEND Recovery routine, then use the **HOOK** command instead of a breakpointing command. The HOOK command sets a special kind of trap (sometimes a JLU) that jumps to Z/XDC logic that examines the local environment to see whether or not it is suitable for Z/XDC processing. If it is not, then steps are take to automagically create the necessary environment (an ESTAEX or FRR routine, as appropriate). Then control is passed to Z/XDC itself so that you can proceed with debugging your program. For more information, see the following:

HELP HOOKS
HELP COMMANDS HOOK

About Breakpoints

Since Z/XDC runs as an ABEND Recovery routine, it can receive control for any ABEND that may occur in the user program, including the s0C1 ABENDs that occur because of the X'00' opcodes used by breakpoints (or the s0D3 ABENDs caused by breakpoints set using the TRAP2 opcode when no TRAP handler has yet been set up).

So when Z/XDC receives control because of one of these ABENDs, it has to perform some extra checking to determine whether the s0C1 is a normal s0C1 or a breakpoint generated s0C1. (The same sort of analysis occurs for s0D3 ABENDs that occur because no TRAP handler has yet been set up.)

In either case, once Z/XDC has decided that the current ABEND is related to breakpointing, it handles it accordingly.

#DIE Macros and DEAD-traps

Breakpoints can be created at user program assembly time via the **#DIE** macro. (Note that the #DIE macro always creates a X'00' opcode; it never creates a TRAP2 instruction.) Or breakpoints can be created via Z/XDC's several breakpointing commands (see below).

Assembled breakpoints created by the #DIE macro are called **dead traps.** Often, the first entry to Z/XDC during a debugging session is the result of a dead trap coded either in the user program or in the **XDCCALL** support program provided as part of the Z/XDC product. During this first call, you would usually use breakpointing commands to set additional traps at those parts of your program that you are interested in debugging.

The **#DIE** macro can be found in the XDCMACS library. The library's factory default name is hlq.XDCV31.XDCMACS. Ask your Systems Programmer for its actual name at your Data Center.

Breakpoint Characteristics

Z/XDC uses breakpoints to set traps and perform execution traces. They can have the following characteristics:

- They can be **transient or persistent.** (See HELP BREAKPOINTS TRANSIENT)
- They can be **active or deferred.** (See HELP BREAKPOINTS DEFERRED)
- They can be **conditional or unconditional.** (See HELP BREAKPOINTS CONDITIONAL)
- They can have **automatic commands** associated with them. (See HELP BREAKPOINTS AUTOCMDS)
- They can be gathered together into **families** that can be manipulated collectively. (See HELP BREAKPOINTS FAMILIES)

Breakpoints can theoretically be set in any address space that the user is allowed to zap see (HELP SECURITY ZAP); however, that usually is not a useful thing to do because such breakpoints will be effective only in those address spaces for which Z/XDC has already been established as part of **the same debugging session** from which the breakpoints are set, and that is rarely the case.

In most cases, such breakpoints, if executed, just cause s0C1 ABENDs. If Z/XDC is present in such an address (and is not part of the current debugging session), then Z/XDC reports the s0C1 as an s0C1 and not as a breakpoint. If Z/XDC is not present in such an address space, then the s0C1 is handled by whatever ABEND Recovery routines might or might not be present.

So what do you do? Well, read on.

Using Hooks to start Debugging Sessions in Other Address Spaces

Breakpoints **cannot** be set into arbitrary other address spaces to create debugging sessions on the fly. If such a breakpoint is set and subsequently executed, then it will be seen in that other address space as just an s0C1 ABEND, not as a breakpoint.

But an "On the fly" debugging session **can** be created in other address spaces using Z/XDC's **HOOK** command. In fact, the HOOK command can be used to insert **Dynamic Hooks** into code located either in the local address space or in a foreign address space. Then when the targeted program's execution reaches the Hook, code is executed that creates a Z/XDC debugging session, and then the program's execution is interrupted to pass control to Z/XDC.

Thus, the **HOOK** command is a very powerful tool for inserting a Z/XDC debugging session "on the fly" into a program without modifying that program ahead of time. The targeted program could be:

- A Request Block driven exit routine (examples: an attention exit, a timer exit, a DCB exit, etc.).
- Another task running in the same address space.
- Any task running in any other address space (as permitted by System security).
- An SRB routine.
- LOCK'd code.
- An FRR protected SRB or task.

For more information, see HELP HOOK.

Additional Topics

The following are the breakpoint commands recognized by Z/XDC. All (except the STEP command) share a common syntax which is described in **HELP COMMANDS SYNTAX BREAKPOINTS.**

AT - Sets traps using persistent breakpoints.
ATX - Sets traps using super persistent breakpoints.
ADEFERRED - Sets persistent deferred breakpoints.
TDEFERRED - Sets transient deferred breakpoints.
STEP - Steps execution through a High Level Language program one language statement at a time.
TRAP - Sets traps using transient breakpoints.
TRACE - Steps execution through machine code in various ways.

For specific information about these commands, type "HELP COMMANDS cmdname" (from the above list).

Other breakpoint related commands:

SET TRACE - Sets various global parameters relating to breakpoints.
SET BREAKPOINTS - Controls various characteristics of specific breakpoints (enabled, disabled, etc.).
LIST TRACE - Displays the breakpoint related parameters that are settable by the SET TRACE command.
LIST BREAKPOINTS - Displays all currently defined active and deferred breakpoints. Also shows current breakpoint related settings and defaults.
OFF - Deletes breakpoints. (This is a shortcut for **SET BREAKPOINTS OFF.)**

Another topic that's of importance to 0C1-type and TRAP2-type breakpoints is **HELP EXECUTIONLEVELS.**

The following commands relate to hooks instead of breakpoints:

HOOK - Creates a Z/XDC debugging session in environments in which Z/XDC has not been previously or properly established. Examples: Code running under other Request Blocks (RBs), in other tasks or in other address spaces.
HDEFERRED - Same as HOOK except that the target location is in a load module that has not yet been loaded into storage.
LIST HOOKS - Displays information about known hooks.
DELETE HOOKS - Removes known and unknown hooks.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

AUTOCMDS - Associating commands for automatic execution.
CONDITIONAL - Conditional vs. unconditional breakpoints.
DEADTRAPS - Assembling breakpoints into your program.
DEFERRED - Deferred vs. active breakpoints.
FAMILIES - Grouping breakpoints into families.

PERSISTENT - See HELP BREAKPOINTS TRANSIENT.
REENTRANT - Setting breakpoints into reentrant programs.
STEPPING-C - Stepping through C program execution in various ways.
TRANSIENT - Transient vs. persistent breakpoints.
TRAPPING - Setting execution traps.
TRACING - Stepping through execution one or several instructions at a time.
TYPES - About breakpoint types (0C1 vs TRAP2).

Help Breakpoints Autocmds

When a breakpoint is reached by user program execution and accepted by Z/XDC, Z/XDC normally automatically issues either of two default storage formatting commands. The choice depends upon the current **TRACE ROLL/SCROLL** setting:

- If **SET TRACE ROLL** is in effect, then the default automatic command is **WHERE.**
- If **SET TRACE SCROLL** is in effect, then the default automatic command is **FORMAT PSW?.**

You may, if you wish, assign your own one or more automatic commands to any specific breakpoint. If you do, then they will be executed instead of the default automatic command when the breakpoint is reached by execution **and accepted** by Z/XDC. To do so, simply enclose your commands within tic marks (') and add them to the end of your breakpointing command. If your commands include tic marks, they will need to be doubled up. Example:

AT .P .T 'F R1? 2 I;TRAP R1? ''ALARM ''''Here'''''''s the problem!''''';GOT'';LIST BREAKPOINTS

OK, here's what's happening here:

- Two persistent breakpoints are created, one at **.P** and the other at **.T.**
- The automatic commands string, **"F R1? 2 I;TRAP R1? 'ALARM ''Here''''s the problem!'';GOT'",** is assigned to **both** of these breakpoints. These commands will be executed every time one or the other of .T or .P is reached by execution.
- The automatic commands contain a nested TRAP command with it's own automatic commands string: **"TRAP R1? 'ALARM ''Here''''s the problem!''''".**
- So when either .T or .P is reached by execution, a new breakpoint will be set at whatever location is then pointed to by R1, and that breakpoint, when it is reached by execution, will sound an alarm that will display the message: **"Here's the problem!".**
- Notice that at each level of string nesting, tic marks need to be (re)doubled.

GENERATING EXECUTION FLOW REPORTS USING AUTOMATIC COMMANDS

Automatic commands can be used to create "batch type" reports (i.e. noninteractive reports) of a program's execution flow or other information. The "report" is the debugging session's log file. To generate the "report", create breakpoints that contain automatic commands for producing the information you want to see. To make the report "automatic", append the LOG and GO commands to the end of the string of automatic commands.

Example: Suppose you want to generate a report to show information about all of the callers of a particular subroutine named "MSGOUT". To do so, you might use the following command:

AT .MSGOUT 'L REGS;L PSW;FORMAT R14! 2;LOG;GO'
The AT command sets a persistent breakpoint at the start of the subroutine named MSGOUT. The remaining commands are not executed until the breakpoint at MSGOUT is executed. They are executed every time MSGOUT is executed. They do the following:

L REGS
This causes a display of all registers.

L PSW
This causes a display of the current PSW.

FORMAT R14! 2
This causes a display of the first couple of lines of MSGOUT's caller.

LOG
This causes all of the generated displays to be immediately written out to the debugging session's log file. The presence of this command is critical, because without it, circumstances can occur where the generated displays are never written out and so are lost.

GO
This causes user program execution to resume. Execution will then proceed normally until either:
- The program ends.
- The program ABENDs.
- The program reaches another breakpoint.

Each time execution reaches **MSGOUT** again, a new report is added to the log file.

USING THE "READ" AND "PROFILE READ" COMMANDS

Any Z/XDC commands may be associated with any breakpoint or WATCH. In particular, **READ** commands may be associated with a breakpoint. If you need to have a particularly long list of commands automatically executed when a breakpoint is reached, then you might place those commands into a file, and associate the appropriate READ command with the breakpoint.

Another useful command to associate with a breakpoint is **"PROFILE READ name".** This command allows you to change the entire windows layout at your terminal's display screen. Thus, you can customize your display to the particular needs of each individual breakpoint. See **HELP PROFILES NAMEDPROFILES EXAMPLE** for an interesting example.

Help Breakpoints Conditional

What are Conditional Breakpoints?

Breakpoints may have conditional expressions associated with them such that when the breakpoint is reached by user program execution, Z/XDC receives control and evaluates the expression. If the result is **FALSE,** then Z/XDC immediately allows the program to resume as if nothing happened, and the breakpoint remains in place regardless of whether it is persistent or transient. (See **HELP BREAKPOINTS TRANSIENT.)**

If the conditional expression evaluates **TRUE,** then user program execution is interrupted, and Z/XDC sends a display to the terminal and awaits commands from the user. Also, if the breakpoint is transient, then it is automatically removed prior to the terminal display being generated.

Both traps **(AT** and **TRAP** commands) and traces **(TRACE** commands) and WATCHs **(TRACE W** commands) may have conditional expressions assigned. In the case of traces, if the conditional resolves as **FALSE,** then the trace point is automatically move to the next stopping point, and user program execution is resumed. Thus, conditional traces have the ability to follow execution until the associated conditional expression (or any WATCH) comes **TRUE.** For more information, see **HELP BREAKPOINTS TRACING CONDITIONAL.**

Z/XDC permits multiple breakpoints (including trace points) to be set at the same location. When multiple **conditional** breakpoints are set to the same address, then the conditional expressions are logically **OR'd:**

- The user program is interrupted if **any** expression resolves **TRUE.**
- User program execution is not interrupted when **all** expressions resolve **FALSE.**

For the **detailed syntax** of conditional expressions, see **HELP COMMANDS SYNTAX BREAKPOINTS CONDITIONS.**

WATCHs

A **WATCH** is a conditional expression that is **not associated** with any specific breakpoint. Instead, it is, in effect, associated globally with all conditional breakpoints.

WATCHs can be a convenient way to define **multiple** conditional expressions that are all tested whenever Z/XDC receives control for any conditional breakpoint.

In other words, when Z/XDC receives control due to a conditional breakpoint, all available conditional expressions are evaluated. This includes:

- All expressions assigned to the one or more breakpoints located at the execution address.
- Plus all WATCHs that currently exist.

Then:

- If **any** of those expressions resolve **TRUE,** then user program execution is interrupted.
- If **all** of those expressions resolve **FALSE,** then user program is silently resumed.

See **HELP BREAKPOINTS CONDITIONAL WATCH** for more information about WATCHs.

FALSE Traps and Traces

As implied above, if you want to set up multiple conditional expressions, WATCHs are a convenient way to do that. But for the most part, a WATCH will be examined only when Z/XDC receives control due to a conditional trap or trace. So you also will have to consider what that conditional expression ought to be as well.

But personally, I find it easier if I separate the process of deciding where to set traps from having to decide what conditionals to assign to them. So typically, I'll just assign **(FALSE)** as the conditional expression for all my traps, and then set up all my actual conditionals to one or more WATCHs.

These are called **FALSE traps.** I find them convenient. For more information, see **HELP COMMANDS SYNTAX BREAKPOINTS CONDITIONS TRUEFALSE.**

Of course, when a particular conditional test is unsuited for universal checking, it remains more appropriate for that test to be associated only with the traps that are set where the test matters.

Types of Conditional Expressions

Generally speaking, there are three kinds of conditional expressions:
- **Value testing** expressions,
- **Countdown** values,
- The boolean constants, **TRUE** and **FALSE** .

With **value testing** conditionals, you can compare the contents of an object against a constant to see if the value is equal to, not equal to, greater than, not greater than (etc.) the constant. The objects that can be compared are:
- The contents of a storage location,
- A register (control, access, or general),
- A PSW,
- A computed numeric value.

Conditional expressions have these characteristics:
- The comparison is always logical, never arithmetic.
- The tested value can be from 1 to 8 bytes wide.
- A mask can be AND'd or OR'd against the value so that individual bits can be selected.

- Expressions can be compounded and nested in the usual ways.

User program execution is interrupted when the comparisons yield a **TRUE** result.

Examples - Testing Registers and Storage

When a value testing conditional expression is used, one component of the conditional expression often is an address expression that is **reevaluated every time** Z/XDC checks the conditional. Thus, the address expression can resolve to a **different storage address** each time the conditional is checked. This can have very powerful consequences.

In the following examples, **R1!+3** and **.BUFPTR!** are instances of address expressions with potentially varying location resolutions.

AT .TOPLOOP (R1+3,AND,01,EQ,01)
User program execution is interrupted at location **.TOPLOOP** when the lo-order bit of general register 1 is on. The breakpoint, being permanent, remains.

AT .TOPLOOP .SUBCALL (R1!+3,GE,0ED8)
Execution is interrupted either at **.TOPLOOP** or **.SUBCALL** when the two bytes starting at three bytes past the storage location pointed to by R1 contain a value that is greater than X'0ED8'. Note that X'FFFF' (-1) is logically greater than X'0ED8'.

T .TOPLOOP (.BUFPTR!,NE,'TYPE 2')
Note that the constant can be given as either a hex string or a character string or both. Also, in this case **T** means **TRAP,** not **TRACE** because the first operand, **.TOPLOOP,** is an address expression (not a trace type keyword).

Examples - Countdowns

If you provide a simple countdown value, you can cause execution to stop when a breakpoint has been executed **n** or more times. Examples:

T .TOPLOOP (30)
User program execution is interrupted at location **.TOPLOOP** on the thirtieth time that **.TOPLOOP** is executed. The breakpoint, being transient, is removed, and Z/XDC awaits commands from the user.

AT .TOPLOOP (30)
User program execution is interrupted at location **.TOPLOOP** on the thirtieth time **and** every time thereafter that .TOPLOOP is executed. The breakpoint, being persistent, is not removed until you issue an appropriate **OFF** command.

Examples - FALSE Traps

The boolean constant **FALSE** is most useful when used on traces and traps that are set in conjunction with **WATCHs.** Example:

AT .TOPLOOP .BADEXIT (FALSE)
T W (R9?+3,EQ,00)
T W (.MYFLAG,AND,40,EQ,40)
T W (1000)
Each time execution reaches either **.TOPLOOP** or **.BADEXIT,** Z/XDC receives control and

performs three tests:
- If **any** one or more of the tests resolve as **TRUE,** the user program is **interrupted.**
- If **all** the tests resolve as **FALSE,** then the user program is **silently resumed.**

The Power of Address Expression Reresolutions

The fact that a conditional expression's address expression is reresolved every time it is used enhances the power of conditionals considerably.

Suppose, for example, the following:
- You need to test some field within a data record;
- However, on each pass through your loop, the data record to be checked can reside in a different buffer, or be a different entry on a queue.
- **R5** always points to the record of current interest.

Then the following conditional breakpoint might be used to search for a particular instance of the data record:

```
T W (536)
T .TOPLOOP (R5?+30,EQ,'PR#'057C)
GO
```

- The **T W** sets a WATCH that limits the number of times the loop will be allowed to iterate.
- The **T .TOPLOOP** creates a breakpoint located at **.TOPLOOP** that checks a field in the data record instance pointed to by **R5** for the value **X'D7D97B057C'.**
- Each time location **.TOPLOOP** is reached by execution, **R5** may contain a different address.
- Therefore, the actual location of the check changes automatically as required by the logic of your program.
- The loop will run either until the specified data record instance is reached, or for 536 iterations, whichever occurs first.
- The **GO** gets it all started.

Testing Computed Numeric Values

Instead of address expressions, value testing conditions can also test computed numeric values.

Some of Z/XDC's **built-in functions** return numeric values, so those built-in functions can be used instead of address expressions for providing testable numeric data. Those functions are:

XADDR(...) - Resolves a given address expression and returns its resolved value as a 4-byte wide number.

XALET(...) - Resolves a given address expression and returns its associated ALET as a 4-byte wide number.

XASID(...) - Returns an ASID as a 2-byte wide number.

For more information, see **HELP COMMANDS SYNTAX BREAKPOINTS CONDITIONS COMPARISONS.**

Example:

AT MYPC.GOTCALLR (XASID(SECONDARY),EQ,007B) 'ALARM'
This can be used in a PC routine to trigger when the calling aspace is the one whose ASID is 007B. All other calling aspaces will be ignored by Z/XDC.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

WATCH - More information about WATCHs.

Help Breakpoints Conditional Watch

What are WATCHs?

A **WATCH** is a special kind of trace definition that has a conditional expression but does not set any breakpoints into the user program's code. Thus, in order for a WATCH to be effective, Z/XDC has to receive control from time to time by other means. Typically, such other means will be one or more traps or traces that have conditional expressions of their own. See **HELP BREAKPOINTS CONDITIONAL** for more information.

WATCHs vs SLIPs

A **WATCH** is **not** an equivalent to a storage alteration PER-type SLIP trap! Unlike SLIP traps, a WATCH will not report an alteration at the moment it happens. Further, a WATCH will never cause Z/XDC to receive control.

WATCHs will report their results only when Z/XDC receives control for **other reasons.** The frequency with which Z/XDC does receive control is related to the number of traps you have set, and the nature of conditional traces you may have running.

For detailed information about the **differences** between SLIP traps and conditional expression driven tracing/trapping (as well as the advantages of each over the other), see **HELP BREAKPOINTS TRACING CONDITIONAL SLIPVSXDC.**

Multiple WATCHs

Multiple WATCHs can exist. Whenever Z/XDC receives control for a trap or trace (such

as T BY), it evaluates the conditional expressions assigned to **all** enabled WATCHs. If **any** expression evaluates **TRUE,** then user program execution is stopped and that WATCH's automatic commands (if any) are processed.

WATCH Evaluation Points

Note that when tracing, Z/XDC will receive control more times than you may think. This is because as Z/XDC is stepping through your program, it has to set **recapture traps** at key points in your code where it has to evaluate whether or not the particular requirements of a given trace command have been met. If they have not, then a new recapture point is chosen, and execution of your program is silently resumed.

For example, for a **T BY** trace, recapture traps need to be placed on each branching instruction as it is encountered. When Z/XDC receives control, it analyses whether or not a branch will actually occur. If not, then the next branch is scanned for, a recapture trap is placed there, and your program is silently resumed.

The point is, **WATCHs** are evaluated at **all** of a trace's recapture points, not just the ones where the trace requirements are met, but also at the ones where requirements are not yet met. This means that a WATCH can trigger at **more places** that you might have expected.

For more information about **recapture traps,** see **HELP BREAKPOINTS TRACING CONDITIONAL.**

Creating, Displaying and Managing WATCHs

A **WATCH** is established by the **TRACE W** command. This command accepts all of the same operands as all other **TRACE, TRAP** and **AT** (etc.) commands. See **HELP COMMANDS SYNTAX BREAKPOINTS** for details.

WATCHs can be disabled, reenabled and purged by the **SET BREAKPOINTS** command. See **HELP COMMANDS SET BREAKPOINTS** for details.

They also can be disabled by an **OFF** command.

They also can be disabled, reenabled and purged via the **X** and **O** shortcut commands issued against a **LIST BREAKPOINTS** display.

Example:

Suppose the following:
- You want to set multiple conditionals to cause execution to stop when **any** conditional is accepted.
- You want all of the conditionals tested at each of several traps.

Then the following is a sample of a set of commands that you might use:

```
T W  (R1,EQ,00000000)        'L REGS;WHERE'
T W  (R4?,NE,'I-CATCHR')     'D R4?'
T W  (.DCBOFLGS,AND,10,EQ,00)
```

TRAP .LOOP1 .ERR .ZPROGRAM (FALSE)

The three **T W** commands create three WATCHs. Each one tests for a different conditional, and two of them have automatic command strings that are to be processed when its WATCH evaluates **TRUE.**

The **TRAP** command sets conditional traps at three locations. However, the **(FALSE)** conditional expression will prevent the traps from ever evaluating **TRUE.** Thus, the traps themselves will not stop program execution. But they do give Z/XDC the opportunity to evaluate the WATCHs, and one of them may eventually interrupt program execution.

So user program execution continues (albeit rather slowly) until at least one of the **WATCH** conditionals resolves **TRUE.** When this happens:

- User program execution is **interrupted.**
- That particular WATCH is **disabled** (but not purged).
- If the WATCH has **automatic commands,** then they are processed.
- (If no accepted WATCH has automatic commands, then the default command, **EWHERE** or **FORMAT EPSW?,** is processed.)
- The debugging session is resumed so that the user can continue to examine his problem.

Eventually, the programmer can, if he wishes, reenable the WATCH simply by:

- Using the **X** shortcut,
  On a **LIST BREAKPOINTS** display.
- Or by issuing a **SET BREAKPOINTS ENABLE** command.

Help Breakpoints DEAdtraps

#DIE is an Assembler Language macro that can be used to assemble a breakpoint into your program. When program execution reaches the code generated by the macro, an s0C1 ABEND occurs that causes control to pass to Z/XDC.

The trap that is generated by the #DIE macro starts with an invalid opcode (X’00’) followed by a 2-byte hex constant that allows Z/XDC to recognize this s0C1 ABEND as being the result of the #DIE macro. The hex constant is X’DEAD’, thus the breakpoints created by #DIE are called "dead traps".

Dead traps can be used to pass either messages or commands to Z/XDC:

- If the X’DEAD’ constant is followed by X’0100’, then following that, Z/XDC expects to find a zero-delimited string of Z/XDC commands (see examples below). If this is the case, then Z/XDC will reset the retry level PSW to point to the halfword boundary past the command string’s delimiting X’00’, and then it will automatically execute the commands contained in the string. The string may contain any one or more Z/XDC commands separated from each other by semicolons

(;). As of this writing (February 2000), the longest command type DEAD trap that Z/XDC can accept and process is 4 kbytes.

- If the X'DEAD' constant is not followed by X'0100', then Z/XDC expects that a message follows. The first byte after the X'DEAD' must be the length of the message, and the message itself must start at the next byte. If this is the case, then Z/XDC will reset the retry level PSW to point to the halfword boundary past the message, and then it will display the message as part of its initial display to the user. Note, the longest message that can be contained in a message type DEAD trap is 255 bytes.

Message Type Dead Trap:

```
         DC    X'00DEAD',AL1(L'msg)
msg      DC    C'message-text'
```

Command Type Dead Trap:

```
         DC    X'00DEAD0100'
         DC    C'Z/XDC commands separated by semicolons'
         DC    X'00'    command string delimiter
```

Subtopics Index

The **#DIE** macro can be used to generate both kinds of dead traps. It can also be used to generate conditional branches to control whether or not a dead trap is executed.

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

#DIE - What the #DIE macro does and how to use it.

Help Breakpoints DEAdtraps #die

The **#DIE** macro generates Z/XDC "dead traps" and conditional branch logic that controls whether or not the dead trap is executed. Macro operands allow you to define:

- The type of dead trap generated: message or command.
- The Z/XDC commands to be executed or the message to be displayed if the dead trap is executed.
- A leading branch instruction to control the conditions under which the dead trap is to be executed or bypassed.

Example: **#DIE L,'SHOULD NOT BE LOW!'**
This generates a message type dead trap that will be executed if the PSW's condition code means "low" (or "minus", or "negative"). If the dead trap is executed, then Z/XDC will receive control and display the message "SHOULD NOT BE LOW!".

Reasons to Use #DIE Macros

#DIE macros, when used properly, can be a great and **convenient** way to detect and control unexpected and unsupported conditions in your program. Example: suppose that you call a subroutine that someone else wrote, and suppose that you have been told that the subroutine will "always" set R15 to a return code of 0, 4, or 8. Well, maybe you should trust this "guarantee", maybe not. If you are skeptical, then you might code the following:

```
    L     R15,=V(HISCODE)
    BALR  R14,R15
    LTR   R15,R15
    #DIE  M            DIE IF NEG.
    CH    R15,=Y(8)
    #DIE  H            DIE IF >8
    B     *+4(R15)
    B     RCIS0
    B     RCIS4
    B     RCIS8
```

Z/XDC itself contains thousands of #DIE macros to detect conditions that I expect should "never" happen. However, in complex environments, such as those that Z/XDC deals with, things that "should never happen" occasionally do happen. With strategically placed #DIE macros, I can trap such events early before their consequences become too complex to debug easily. Example: In the early 1980s, IBM increased the maximum return code from the TGET macro from 20 to 28. After my TGET macro, I coded a branch vector table to jump according to the return code. Not knowing ahead of time that IBM was going to increase the possible RC, I did not have slots in my branch vector for codes 24 and 28. Fortunately, I had code similar to the preceding example to check for and abort if I received an RC greater than 20. Eventually, when IBM increased the possible RC, my dead trap was executed, and execution was stopped immediately. If I did not have the dead trap, a wild branch would have occurred, and the real problem would have been much more difficult to debug!

One very good use for #DIE macros is to guarantee the correctness of interfaces to or from other routines.

Syntax:

```
    name #DIE mnemonic,mi-operands,...,'msg',TYPE=MSG    ,RI=YES
                                       'cmds',TYPE=CMD   ,RI=NO
```

Rules:

- Any necessary number of **mi-operands** may be given.
- All operands are optional.
- Omitted operands do **not** need to be indicated by leading commas.

Operands:

name
This may be a standard statement label. If given, then it is used to label the first executable statement generated by the macro.

mnemonic
This must be the mnemonic **(excluding** the starting **B** or **J)** of any RX-type, RS-type or RI-type branching machine instruction. Examples:

```
    NE      instead of BNE  or JNE
    XLE     instead of BXLE or JXLE
    CT      instead of BCT  or JCT
    omitted instead of B    or J
    N       instead of NOP  or JNOP (i.e. Never)
    etc.
```

This operand specifies either the condition under which the dead trap **is** to be executed or the branch instruction whose target is to be the dead trap.

mi-operands,...
If the branch instruction specified by the given mnemonic requires one or more operands **in addition to** the branch target address, then they must be given, delimited by commas, following the mnemonic. Examples:

```
    #DIE C,15,'WHOOPS!'
    #DIE XH,R1,R4,'OH-OH, MISSED!'
    #DIE CT,R2,'HEY! I THOUGHT R2 WAS GONNA BE SET TO 1.'
    #DIE NE,'WHA-DA-YA MEAN! NOT EQUAL?'
```

The #DIE macro uses the given (or default) message to form a dead trap and to make it the branch target address.

msg
msg,TYPE=MSG
TYPE=MSG indicates that a "message type" dead trap is to be generated. Specifying **TYPE=MSG,** however, is optional since this is the default type. **Msg** is the message text to be displayed by Z/XDC if the dead trap is executed. It too is optional. If omitted, then a default message is generated. If given, then it must be enclosed within tic marks ('). Embedded tic marks and embedded ampersands (&) must be doubled. Example:

```
    #DIE NL,'I''M DEAD && GONE NOW!'
```

cmds,TYPE=CMD
TYPE=CMD indicates that a "command type" dead trap is to be generated. **Cmds** is the string of commands to be executed by Z/XDC if the dead trap is executed. It must be enclosed within tic marks ('). Embedded tic marks and embedded ampersands (&) must be doubled. Individual Z/XDC commands within the string must be separated from each other by semicolons (;). The string's trailing delimiter need **not** be given; the macro will append the necessary X'00'.

The command string may contain embedded comments, but be careful that intended commands following the comment string are not accidentally treated as being a part of the comment! This arises because, within comment strings, Z/XDC's command delimiter character, the semicolon **(;)** can sometimes be interpreted as text instead of as a delimiter. The trick is to be careful about what follows the semicolon:

- If the immediately following character is either an asterisk **(;*)** or an alphabetic character **(;commandverb),** then the semicolon ends the comment, and either a new comment (for **;*)** or a new command (for **;commandverb)** is started.

- On the other hand, if the semicolon is directly followed by any other character (by a blank, by a digit, by whatever), then it is considered to be just part of the commentary text.

For more information, see **HELP COMMANDS ASTERISK.**

Example: **#DIE E,'*** Oh-oh; R1 is screwy! ***;F R1? 5;WHERE',TYPE=CMD**
When the **#DIE** macro is executed, if the condition code implies EQUAL or ZERO, then Z/XDC accepts control and processes the following commands:

***** Oh-oh; R1 is screwy! *****
F R1? F
WHERE

RI=YES
RI=NO
omitted
When the branching instruction to be generated by the macro is an extended mnemonic (BNE, JZ and such), this operand controls whether the branching instruction generated to jump to (or around) the dead trap is a base-displacement type instruction (Bxx) or a relative-immediate type instruction (Jxx):.

RI=YES
The #DIE macro will generate a Jxx type branching instruction.

RI=NO
The #DIE macro will generate a Bxx type branching instruction.

omitted
The #DIE macro will generate Bxx or Jxx instructions according to the ARCHLVL= operand of the most recent prior SYSSTATE macro:
- When ARCHLVL=0, Bxx instructions will be generated.
- When ARCHLVL>=1, Jxx instructions will be generated.

SYSSTATE is an IBM macro that can be used to make other IBM and user macros aware of various environmental characteristics in which the assembled code is intended to execute. For more information see IBM's manual: "z/OS MVS Programming: Assembler Services Reference, Volume 2 (IARR2V-XCTLX)" (SA22-7607)

Generating Inline vs. Remote Traps

If the #DIE macro is given with one or more "mi-operands" fields given, then the dead trap will be generated remotely. It will be generated as an Assembler Language literal referenced by the address portion of the generated branch or jump instruction. Examples:

```
 EXAMPLE1 #DIE  C,4
+EXAMPLE1 BC    4,=X'00DEAD04F0F0F0F1'

 EXAMPLE2 #DIE  XH,R3,R0,'WHOOPS'
+EXAMPLE2 BXH   R3,R0,=X'00DEAD06E6C8D6D6D7E2'
```

On the other hand, if the #DIE macro is given without any "mi-operands" fields, then

the dead trap is generated inline, and the branch instruction generated by the macro has a "polarity" that is opposite to that specified by the "mnemonic" field. Examples:

```
 EXAMPLE3 #DIE  M
+EXAMPLE3 BNM   SKIP0003
+         DC    X'00DEAD',AL1(L'DEAD0003)
+DEAD0003 DC    C'0001'
+SKIP0003 DS    0H

 EXAMPLE4 #DIE  NZ,'WHOOPS'
+EXAMPLE4 BZ    SKIP0004
+         DC    X'00DEAD',AL1(L'DEAD0004)
+DEAD0004 DC    C'WHOOPS'
+SKIP0004 DS    0H

 EXAMPLE5 #DIE  'WHOOPS'
+EXAMPLE5 NOP   SKIP0005
+         DC    X'00DEAD',AL1(L'DEAD0005)
+DEAD0005 DC    C'WHOOPS'
+SKIP0005 DS    0H
```

Regardless of whether the macro generates the dead trap inline or out of line as a literal, you must **always** choose the mnemonic operand based on the thought that if the intended branch occurs, then the dead trap will be executed. This subtlety is clearly illustrated by examples 1 and 3 above. **Example 1** generates an inline dead trap while **example 3** generates an out of line literal. However, the two examples accomplish the **identical** logical result: The dead trap is executed if the PSW's condition code is 1 (meaning, "the result is low, minus, or negative").

TRACE BY's Special Support for Inline DEAD Traps

Note, the **TRACE BY** command has special support for inline **#DIE** traps. because inline traps are preceded by a conditional branch around them, a **BY** type trace would normally stop at that branch whenever the tested condition was FALSE (ie, when "everything is OK" and, therefore, the branch around was TRUE). Such stops wind up being unnecessary and just a nuisance. So the **TRACE BY** support contains special logic that allows execution to just continue when it detects a successful branch that is only bypassing an inline DEAD Trap. For more information, see HELP COMMANDS SET TRACE IGNORE.

Additional Information

The #DIE macro is distributed with the Z/XDC product. It may be found in the **XDCMACS** library. The library's factory default name is hlq.XDCV31.XDCMACS. Ask your Systems Programmer for its actual name at your Data Center.

For additional information, please review the comprehensive commentary to be found within the macro.

Help Breakpoints DEFerred

Z/XDC breakpoints (and hooks) can be either "active" or "deferred". An active breakpoint is one that has been set into a program that is currently resident in storage. A deferred breakpoint is one that is not yet set. Deferred breakpoints are targeted towards load modules and program objects that are not yet resident in storage. They will not be set until the target load module or program object is loaded into private storage by z/OS.

Z/XDC also supports deferred hooks. Like deferred breakpoints, these are hooks that are targeted towards load modules (or program objects) that will be brought into storage in the future. For more information about deferred hooks, see HELP HOOKS DEFERRED.

When a deferred breakpoint is created, Z/XDC monitors all subsequent LOAD, LINK, ATTACH, and XCTL macros issued by your programs. If any of these macros should cause a targeted module or program object to be loaded into storage, then during the load process, an active copy of the deferred breakpoint is set into the module or object at the designated offset. All of the deferred breakpoint's attributes are copied when creating the active breakpoint. These include:

- Its transient vs. persistent attribute.
- Its automatic commands, if any.
- Its conditional expression, if any.
- Its family identifier.

Note that the deferred breakpoint definition continues to exist in case another copy of the targeted module or object is loaded again later. Accordingly, any deferred breakpoint can lead to the creation of one or more active breakpoints, depending upon how many physical copies of a targeted module or object are loaded from disk. This creation process is called "cloning": Active breakpoints are "cloned" from a deferred breakpoint whenever a targeted module or object is read from disk.

Deferred Breakpoints and Security

Deferred breakpoints cannot be used to violate security. When a targeted module or object is loaded into storage, Z/XDC checks to ensure either that the user program (at the time the xDEFERRED command was issued) was in supervisor state, or that the module's or object's storage key matches the execution key of the user program. Thus, for example, a problem mode program can successfully target only modules or objects that are loaded into key 8 or key 9 storage.

Deferred Breakpoints and Reentrant Load Modules and Program Objects

Please note that reentrant programs from authorized libraries are loaded into key 0 storage. Thus, if a problem mode program attempts to target a deferred breakpoint towards such a module or object being loaded from such a library, then the attempt will fail. The easiest way to circumvent this problem is to concatenate the authorized library with **any** non-authorized library and allocate that concatenation to your program's JOBLIB or STEPLIB (or TASKLIB if the debugging session has been started via Z/XDC's Startup Panel in ISPF or via the XDCCALL program or any of its aliases). This is known as "poisoning the concatenation".

Deferred Breakpoints and Refreshable load modules and program objects

On systems that have the REFRPROT facility both installed and activated, programs that are refreshable (REFR) are loaded by the system into key 0 storage, regardless of the load library's authorization level and regardless of the RENT attribute. Therefore, problem state programs cannot place deferred breakpoints (or hooks) into such programs without first turning REFRPROT off. For more information about this problem, see HELP DEBUGGING REFRESHABLE.

Deferred Breakpoints and Modules Already Present in Storage

A deferred breakpoint **never** clones an active breakpoint into copies of a target module or program object that are already resident in storage. This means:

- If a copy of a target module or object is already in storage at the time that an ADEFERRED or TDEFERRED is issued, the resident copy remains unaffected. If you need to place an active breakpoint into that copy, then you will also have to use the AT, ATX, or TRAP command, as appropriate.

- If a target module or object has the "RENT" or "REUS" Binder attributes, then the System may satisfy LOAD, LINK, ATTACH, and XCTL requests by returning a pointer to a copy of the target module or object that is already in storage due to a previous request, thus the System may decide to avoid reading a new copy of the module or object into storage. When this occurs, Z/XDC's code that clones active breakpoints from deferred ones is bypassed, and so cloning does not occur. Cloning occurs only when a load module or program object is actually read from disk.

Targeting a Deferred Breakpoint

A deferred breakpoint can be targeted towards:
- A load module's or program object's entry point.
- Its load point.
- The start of any csect or element within the module or program object.
- Or any offset relative to any of the above, as long as the resulting location falls within the physical limits of the load module or program object.

To refer to the module's or object's entry point, just specify the its name.
Examples:

```
AD MYMODULE
TD MYMODULE+3DC
TD MYMODULE-1C   - (valid if the object's entry point is not located at its load
                   point)
```

If a load module's or program object's entry point is not located at its load point, and if you want to target a deferred breakpoint relative to the physical start instead of to its entry point, then just use Z/XDC's syntax for referring to an object's extents. Examples:

```
AD MYMODULE.X#1
AD MYMODULE.X#1+35C
```

If you wish to target a deferred breakpoint relative to a csect or element within

your load module or program object, then you must first make the object's module map available to Z/XDC so that it has a way to determine the csect/element locations within the module/object. To do this you must use Z/XDC's **DMAP** command to read the module's or object's module map as if it were a dsect. Example:

```
    DMAP MYMODULE.
```

The trailing **dot** (.) is important! It is the presence of this dot that informs Z/XDC that you want the DMAP command to read a module map.

Once you have read the module map, you can then reference the module/object's csects/elements with the ADEFERRED, TDEFERRED, and HDEFERRED commands. You do **not** need to issue the USING command to place the map anywhere. It is sufficient that the map has been read, it does not need to be assigned to storage. Examples:

```
   DM MYMODULE.
```



```
   TD MYMODULE.MYCSECT
```

```
   AD MYMODULE.MYCSECT+A0
   AD MYMODULE.MYCSECT-3C
```

Transient Deferred Breakpoints

When an active breakpoint is cloned from a deferred breakpoint, the active breakpoint will be transient or persistent according to whether the deferred breakpoint was created by the TDEFERRED or ADEFERRED command.

A transient deferred breakpoint is not automatically deleted until one of the active breakpoints that has been cloned from it is reached by user program execution. Thus, although generally not common, it nonetheless is possible for a transient deferred breakpoint to clone more than one active breakpoint before it is deleted.

- For modules and objects loaded via LINK, ATTACH, and XCTL macros, multiple clonings of transient deferred breakpoints is unlikely because generally, the first cloned active breakpoint will be executed almost immediately after the cloning has occurred. However, multiple clonings can occur if the cloned active breakpoint is placed at a location that failed to be executed thus permitting the deferred breakpoint to survive for subsequent loads (if any) of the same module or object into storage.

- For modules and objects loaded via a LOAD macro, multiple clonings of transient deferred breakpoints can occur under the following circumstances:
  - Your program issues multiple LOAD macros against the same targeted load module or program object.
  - The module/object is neither RENT nor REUS, so each LOAD causes a separate copy to be read into storage.
  - Program execution does not reach any of the cloned active breakpoints until after several LOADs have occurred.

Miscellaneous

Deferred breakpoints in overlay modules are not supported anywhere but in the root segment.

Deferred breakpoints are created by the following commands:

ADEFERRED - Sets persistent deferred breakpoints.

TDEFERRED - Sets transient deferred breakpoints.
HDEFERRED - Sets deferred hooks.

The following commands all set active breakpoints:

AT - Sets traps using persistent breakpoints.
ATX - Sets traps using super persistent breakpoints.
TRAP - Sets traps using transient breakpoints.
TRACE - Performs various execution traces.
HOOK - Sets hooks.

The above breakpoint commands (but not the hooking commands) all share a common syntax which is described in HELP COMMANDS SYNTAX BREAKPOINTS.

Like active breakpoints, deferred breakpoints (but not deferred hooks) can have conditional expressions and automatic commands associated with them. These expressions and commands do not, however, affect the function of the deferred breakpoint in any way. Instead, they are simply inherited by the active breakpoints that are cloned from them. Thus, automatic commands and conditional expressions have no effect until a resulting active breakpoint is reached by user program execution.

Help Breakpoints Families

Multiple breakpoints can be grouped together into **families** of related breakpoints. The primary properties of breakpoint families are the following:

- When any active breakpoint in a family is reached by execution and accepted by Z/XDC, then all **transient** members of that family are automatically deleted. Note that it is possible for a family to include both persistent and transient breakpoints.

- The **OFF** command (or the **SET BREAKPOINTS OFF** command) can select breakpoints according to many different categories. Membership in specified families is one of those selectable categories.

A breakpoint's family identifier is a decimal number ranging from 1 to 999. A breakpoint's label includes its family id number. If the labels of two different breakpoints contain the same decimal number, then those two breakpoints are members of the same family. A breakpoint's label can be displayed by storage formatting commands (FORMAT, WHERE, SHOW, or FIND) and by the LIST BREAKPOINTS command. Example:

```
FORMAT PSW?
AT00003B STM   R14,R12,C(R13)
AT00002C LR    R1,R13
TR00002D LA    R13,34(,R15)
          ...
```

- Breakpoints AT00002C and TR00002D are in the same family (family #2).
- Breakpoint AT00003B is in a different family.
- As soon as either AT00002C or TR00002D is reached by user program execution, breakpoint TR00002D will be automatically deleted.
- The execution of breakpoint AT00003B will have no effect on TR00002D.

A breakpoint's family is established in any of several ways:

- By default, a breakpoint command assigns the breakpoints that it creates to a family that is different from breakpoints created by prior commands. This default can be overridden by appropriate command syntax. Examples:

  T R14! R14!+4 .SYNAD
  This command creates three transient breakpoints. They are all assigned to a single, new family. This means that if execution should reach any of these three breakpoints, then Z/XDC will automatically delete all of them from your program.

  AT =3,.TOPLOOP .TOPLOOP2
  This command creates two persistent breakpoints and assigns them to family number 3. It may or may not be the case that previously created breakpoints already exist in family number 3.

  TRAP =,.NOTHERLP
  The one breakpoint created by this command is assigned to the same family used by the most recently issued preceding breakpoint command (in this example, to family #3).

- As illustrated by the first of the preceding examples, if a single breakpointing command creates multiple breakpoints, then they all are assigned to the same family. For a practical example, see **HELP BREAKPOINTS TRACING SUBROUTINES.**

- When multiple **T** and **A** shortcut commands are typed at the left of a fullscreen display such that they are all executed via a single press of the ENTER key, then all resulting breakpoints are assigned to a single family, even when a mixture of **T** and **A** commands is used. For more information, see HELP SHORTCUTCMDS.

- All active breakpoints cloned from a single deferred breakpoint are assigned to the same family to which the deferred breakpoint is assigned.

- Families do not apply to hooks, only breakpoints.

Help Breakpoints Persistent

For complete information about persistent breakpoints, see HELP BREAKPOINTS TRANSIENT.

Help Breakpoints Reentrant

When you issue a breakpointing command, Z/XDC creates the breakpoint by locating the targeted machine instruction and changing its opcode from whatever it was to X’00’. This, of course, requires that the target machine instruction reside in storage that is not "store protected" with respect to Z/XDC. For an explanation of storage protection, see HELP DEBUGGING REENTRANT STORAGEKEYS.

Generally, whenever z/OS initiates a program (authorized or non-authorized), it

assigns that program to use execution key 8, and it sets the default storage key for its GETMAINs to be "storage key 8, fetch protected". Thus, as one would expect, a program is allowed to fetch from and store into storage that it owns.

However, when a program is marked during linkedit processing as being either reentrant (RENT) or refreshable (REFR), then the rules may change, and if they do change, then they will change in differing ways depending upon the exact details if the situation.

In addition, the program's **own** storage (i.e. the storage containing the code itself) generally is also assigned to fetch protected key 8. This means that the program can both read from **and write into** its own code and assembled data areas.

However, if a program is marked either reentrant (RENT) or refreshable (REFR), then when it is loaded into the storage, z/OS may load it into fetch protected key 8 storage, **or** it may load the program into fetchable key 0 storage! If the latter, then the program can read its own code (because its storage is fetchable), but it cannot overwrite itself (because the storage key is 0 while the execution key is 8).

Normally, this does not present any execution problem because by definition reentrant programs and refreshable programs are not supposed to store into themselves anyway. **But** it does, under certain circumstances, make it difficult to use Z/XDC's breakpointing and zapping commands.

When Z/XDC is running authorized, the possibility that the program being debugged may reside in key 0 storage does not matter because being authorized, Z/XDC will have the ability to overcome all forms of storage protection. However, when Z/XDC is running **non-authorized,** steps must be taken to force programs into user key storage; otherwise, setting breakpoints, stepping through code (tracing) and zapping program code will be impossible!

The issues presented by reentrant and refreshable programs, and the techniques for dealing with then are discussed in detail in the following topics:

HELP DEBUGGING REENTRANT
HELP DEBUGGING REFRESHABLE

Please read those topics for more information.

Help Breakpoints Stepping-c

Stepping is the processing of allowing program execution to "step through" code one or a few instructions or statements at a time. At each step, Z/XDC stops execution and gives you, the user, the chance to see what has happened.

A **step** may be as short as a single machine instruction or language statement. Or it may be as long as an entire subroutine that you wish to let run without examination.

- For Assembler, stepping is called **tracing** and is done via the **TRACE** command.
- While for C programs, the **STEP** command is used.

Subtopics Index

Stepping through C programs is discussed further in the following topic. Type

an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

C - Using the **STEP** command to step execution through C program statements.

Help Breakpoints Stepping-c C

The following information pertains to the c/XDC Licensed Feature. See **HELP SUPPORT FEATURESANDCAPS** for more information.

What is Stepping?

When execution is stopped at some High Level Language statement (such as an XL C/C++ and Metal C statement), **stepping** is the processing of allowing that statement to execute and then stopping at some other statement. Usually, that "some other" statement is the next following statement, but there are variations.

To step execution through an HLL program (such as XL C/C++ and Metal C), you use the **STEP** command. The STEP command can be used only with High Level Languages; it cannot be used with machine code. (To "step" execution in Assembler, you use the **TRACE** command. See **HELP BREAKPOINTS TRACING** for more information about that.)

There are four variations of the **STEP** command:

- **STEP IN:** When execution is poised on a statement that is about to invoke a function or subroutine, **STEP IN** causes stepping to follow execution into the subroutine.

- **STEP OUT:** When execution is within a subroutine, **STEP OUT** allows the subroutine to run to completion, and execution is recaptured upon return to the calling routine.

- **STEP OVER:** When execution is poised on a statement that is about to invoke a function or subroutine, **STEP OVER** allows the subroutine to execute. Execution is recaptured upon return to the calling routine.

- **STEP:** When no operand is given, the **STEP** command behaves like **STEP IN** or **STEP OVER** according to the current **SET STEP ACTION=** setting. See **HELP COMMANDS SET STEP** for more information.

When execution is on any statement that does not invoke a function or subroutine, both **STEP IN** and **STEP OVER** behave identically.
- They both simply allow the current statement to execute, and they recapture execution prior to the execution of the next following statement.
- They differ only when a subroutine or function is about to be invoked.

Subtopics Index

The following additional information is available. Type an **H** at the left to select

directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

DEFAULTSTEP - Using STEP without operands
PFKEYS - STEP'ing using PF keys
MIXWITHTRACE - Intermixing TRACE'ing with STEP'ing
MACHINECODE - Showing machine code in High Level Language (HLL) programs.
AUTOSTEP - Automatically STEP'ing past prologs and epilogs
MAPSREQUIRED - STEP'ing requires DWARF based source image maps

Help Breakpoints Stepping-c C Defaultstep

When the **STEP** command is given without operands, the Factory Default is that it works like **STEP OVER.** However, you can use the **SET STEP** command to change it to behave like **STEP IN** instead.

But why would you want to do that? **Safety,** that's why. Consider:

- Suppose you have STEP set to STEP OVER, and you're stepping along, and stepping along, and suddenly you notice that the statement you stepped over a couple of steps back called a subroutine that you meant to step through in detail. So what do you do now? Nothing. It's too late. you can't go back. All you can do is start over. [sigh]

- Now consider if you had STEP IN set as your default. And now you're stepping along, and stepping along, and suddenly you notice that you're in a subroutine that you did not intend to step through. So what do you do now? Easy, you STEP OUT and continue on your way. No muss, no fuss, no starting over. It's all good.

Note: STEP IN and STEP OVER behave identically for all C statements other than those that invoke functions. So for most statements, it simply doesn't matter whether STEP behaves like STEP IN or like STEP OVER. It only matters (as described above) for implied or explicit function calls.

Help Breakpoints Stepping-c C Pfkeys

Z/XDC's factory default PF keys for High level Language programs include three definitions that are particularly useful for stepping. (For complete information about Z/XDC's PF key settings, see **HELP FULLSCREEN PFKEYS DEFAULTKEYS.)** These definitions are:

- **F9: STEP IN**
- **F10: STEP OUT**
- **F11: STEP OVER**

The above described settings for PF keys **F9, F10** and **F11** are well suited for High Level Language (HLL) debugging; however, they cannot be used for Assembler debugging.

So when Z/XDC detects that the program being debugged is **not** a High Level Language program, different PF keys are loaded that use **TRACE** commands instead of **STEP.** (For more information, see **HELP BREAKPOINTS TRACING PFKEYS.)**

To load a factory default profile designed for **C** programmers, issue **PROFILE RESET C.** For more information, see:

- **HELP PROFILES FACTORYDEFAULTS CWIDE**
- **HELP COMMANDS PROFILE RESET**

As you may have noticed, the three **STEP'ing** PF key definitions supplant the normal settings of **SWAP, RIGHT** and **LEFT** for PF keys **F9, F10,** and **F11. SWAP, RIGHT** and **LEFT** are, however, still useful in Z/XDC, so these definitions are still available via the **shift-F9, shift-F10** and **shift-F11** keys.

Help Breakpoints Stepping-c C MIxwithtrace

As noted elsewhere, the **STEP** command is oriented towards stepping execution through High Level Language programs, while the **TRACE** command is geared towards stepping through machine code. So typically, a C programmer would likely want to use **STEP** exclusively and just leave **TRACE** alone.

But a programmer who knows both C and Assembler... he might well want to intermix the two, and in Z/XDC, that's completely supported. Use the **STEP** command to move execution to (usually) the next High Level Language statement. Use the **TRACE** command to step through machine code without regard to where the HLL statement boundaries are.

If you're going to use the **TRACE** command within an HLL program, you're probably going to want to see the machine code underlying HLL statements. That's discussed in **HELP BREAKPOINTS STEPPING-C C MACHINECODE.**

Help Breakpoints Stepping-c C MAChinecode

When you're debugging a High Level Language program (such as XL C/C++ and Metal C), you generally do not want to see machine code in your displays. But on the other hand, if you've decided that you need to use TRACE commands, then I guess you do want to see the machine code after all. That will happen automagically whenever the next instruction to be executed is not the first machine instruction behind an HLL statement.

But if that's not good enough, the FORMAT and WHERE commands have operands that will allow you to intentionally display the machine code whenever you want.

- FORMAT [...] **BOTH** displays storage showing the machine code intermixed with the HLL statements.

- FORMAT [...] **CURRENTMCODE** displays the machine code behind only the current HLL statement (i.e. the statement at or into which the PSW points). For all other statements, the machine code is suppressed.

The Factory Defaults for the above settings are **SOURCE** and **CURRENTMCODE,** respectively. You can use the **SET FORMAT** command (or the **PROFILE** command) to change these defaults.

You can use the **LIST FORMAT** command (or the **PROFILE** command) to display the current settings.

The current format settings can be saved in (and reloaded from) session profiles. It also can be shown and changed in the Profile Menuing System. See HELP PROFILES for more information.

Help Breakpoints Stepping-c C Autostep

When a programmer’s XL C/C++ and Metal C program is loaded into memory for execution, the programmer’s logic will be preceded by runtime setup code that builds the execution environment within which the program’s logic runs. Typically, while an Assembler programmer *might* want to step through this logic, a C programmer... not so much.

When execution reaches and begins to execute a C program, **autostepping** is what we call the automatic process of letting execution run through runtime setup code and not stopping until it reaches the machine code underlying the program’s first C statement.

A C programmer would want autostepping to be on. An Assembler/C programmer once in a while might want it to be off. So c/XDC has commands that allow the programmer to control this.

- You can use the **SET STEP** command (or the **PROFILE** command) to turn autostepping on or off.
- You can use the **LIST STEP** command (or the **PROFILE** command) to show autostep’s current setting.

The current autostep setting can be saved in (and reloaded from) session profiles. It also can be shown and changed in the Profile Menuing System. See HELP PROFILES for more information.

Help Breakpoints Stepping-c C MAPsrequired

It is a requirement of the **STEP** command that the HLL program to be STEP’d through be mapped, i.e. that the program’s DWARF based source image map be loaded and applied to the memory in which the program resides. For more information about such maps, see **HELP MAPS DWARF.**

When **AUTOMAPing** is turned on, such maps are automatically loaded by Z/XDC whenever Z/XDC detects that execution has reached a program for the first time. For more information, see **HELP MAPS AUTOMAPPING.**

But of course, not all code within a C load module has maps or is even mappable. For example runtime support routines are not mappable. Or DWARF data simply might be absent for programs included from multiple compilations. Or some sections of a load module might have been written in other programming languages not supported by Z/XDC. Or whatever.

So what does **STEP IN** do when the called function does not have a source image map? It doesn't step in; it steps over, which usually is the right thing to do.

However, if you really do want to follow execution into an unmapped module, you can do so using **TRAP** and **TRACE** commands instead of **STEP.**

Help Breakpoints TRANsient

Z/XDC breakpoints can be either **transient** or **persistent.** Persistent breakpoints also can be either **normal** persistent or **super** persistent.

- A transient breakpoint is a **one hit** breakpoint. If a user program executes a transient breakpoint, then when Z/XDC receives control, one of the first things that it does is automatically purges or disables the breakpoint, thus restoring the original machine instruction opcode to the breakpointed location. (A transient breakpoint also, of course, can be purged at any time by Z/XDC's **OFF** command or **SET BREAKPOINTS OFF** command.)

  Z/XDC chooses between purging or disabling a transient breakpoint according to whether the transient breakpoint definition is considered to be trivial or complex:
  - If the definition has an automatic command or a conditional expression, then the transient breakpoint is complex.
  - Otherwise, the transient breakpoint is trivial.

  Complex transient breakpoints are disabled, rather than removed. But this action can be overridden by the **TRAP** (or **TDEFERRED)** command's **REMOVAL=DISABLE** or **=PURGE** operand. See **HELP COMMANDS SYNTAX BREAKPOINTS** for the details.

  If a breakpoint (of any type) has been reached by user program execution, and it is a member of a **family** of breakpoints, at least some of which are transient, then Z/XDC automatically disables or purges all transient breakpoints in that family.

- A **persistent** breakpoint can be removed (i.e. purged or disabled) only by an **OFF** command or a **SET BREAKPOINTS OFF/DISABLE** command. Whether or not a persistent breakpoint is ever executed does not matter: It continues to exist until an appropriate **OFF** or **SET BREAKPOINTS OFF/DISABLE** command is issued.

  If the persistent breakpoint is a member of a family that includes transient breakpoints, then the transient breakpoints in the family will be purged or disabled if **any** breakpoint in the family (including a persistent breakpoint!) is executed.

- A **super persistent** breakpoint (set by the **ATX** command) is a persistent breakpoint that can be removed only by explicit and by-type references. When an **OFF** command or **SET BREAKPOINTS OFF/DISABLE** command uses a general reference, ATX-type breakpoints are not removed.

  - When an **OFF** command specifically names an **ATX** breakpoint (example: **OFF ATX0005A),** that's one way they can be removed.

  - Address directed **OFF** commands and the **O** shortcut can remove **ATX** breakpoints also, but only if they are the **only breakpoints** at the targeted location. If other breakpoints exist there, those will be removed, but the ATX breakpoint will be left behind. So if you really to want to remove the ATX breakpoint as well, then you may have to issue the **OFF addressexpression** command (or **O** shortcut) twice.

Transient breakpoints are one-hit breakpoints that are removed as soon as they are reached by execution. They also are removed when other breakpoints (regardless of type) that are members of the same family are reached by execution.

Transient breakpoints are used or set by the following commands:

STEP - Steps execution through High Level Language programs (such as XL C/C++ and Metal C) one language statement at a time.
TRAP - Sets traps using transient breakpoints.
TRACE - Steps execution, in various ways, through machine code.
TDEFERRED - Sets deferred transient traps.

Persistent breakpoints remain in place until they are removed by an **OFF** command. They are set by the following commands:

AT - Sets traps using persistent breakpoints.
ADEFERRED - Sets deferred persistent ATs.

Super persistent breakpoints remain in place until they are **explicitly** or **categorically** referenced by an **OFF** or **SET BREAKPOINTS OFF/DISABLE** command. They are set by...

ATX - Sets traps using super persistent ATs.

Hooks, like transient breakpoints, are one-hit traps that are removed immediately upon being reached by program execution. However, unlike all other breakpoints, hooks have the ability to **create debugging sessions** on the fly. (All other breakpoints require that a debugging session already preexist.)

Hooks are set by the following commands:

HOOK - Sets a hook at one or more given locations.
HDEFERRED - Sets a hook into a load module that has not yet been LOAD'd into storage.

The above breakpoint commands all (except **HOOK** and **STEP)** share a common syntax which

is described in **HELP COMMANDS SYNTAX BREAKPOINTS.**

For the other commands, see:
- **HELP COMMANDS HOOK**
- **HELP COMMANDS STEP**

Help Breakpoints TRAPping

What is Trapping?

Z/XDC's execution trapping support allows you to interrupt your program's execution at any point you wish. Simply issue the appropriate trapping command specifying the one or more locations at which you want the trap(s) placed. Z/XDC will place the trap(s) at the specified location(s) and then await further commands. Eventually, you should use the **GO** command to cause Z/XDC to relinquish execution thereby allowing your program to resume. When your program reaches one of your traps, an interrupt will occur, your program will stop at that point, and Z/XDC will regain control and await your commands.

The following are the trap setting commands recognized by Z/XDC. They mostly share a common syntax which is described in **HELP COMMANDS SYNTAX BREAKPOINTS.**

AT - Sets traps using **persistent** breakpoints.
ATX - Sets traps using **super persistent** breakpoints.
TRAP - Sets traps using **transient** breakpoints.

ADEFERRED - Sets **deferred persistent** breakpoints for load modules not yet in storage.
TDEFERRED - Sets **deferred transient** breakpoints for load modules not yet in storage.

HOOK - Sets **hooks** into locations whose local environments are **not yet set up** for Z/XDC processing.
HDEFERRED - Same as **HOOK** except the target load module is not yet in storage.

Hooking vs. Trapping

Hooks and **Traps** both are methods for capturing execution from a running program and passing control to Z/XDC. However, that's where the similarity ends. Their purposes are are quite different, and their two mechanisms are worlds apart!

The underlying mechanism for traps are called **breakpoints.** The underlying mechanism for hooks are call, well, **hooks:** The **key differences** between hooks and traps are these:

- For **traps,** a Z/XDC debugging environment **must already exist!** This is because under the covers, breakpoints are invalid opcodes **(X'00')** that cause code-0001 program interrupts. In order for these interrupts to be intercepted and properly

interpreted as breakpoints, Z/XDC must have already been established as your running program's **newest** recovery routine (ESTAE[X] usually).

- For **hooks,** on the other hand, a preexisting debugging session is **not** necessary. The whole point of a hook is to **create a debugging session** where none existed. So under the covers, a hook is a dynamically created **JLU** machine instruction (usually) that jumps execution to a rather complex service routine whose job is to set up Z/XDC as the newest ESTAEX (or FRR) for the currently running program.

For further information about hooks, see:

- **HELP HOOKS**

- **HELP COMMANDS HOOK**

- **HELP COMMANDS HDEFERRED**

Setting Traps

All of the breakpointing commands (i.e. the **trapping** commands, not the HOOK and HDEFERRED commands) can be given **multiple** address expressions indicating that multiple breakpoints are to be set.

The address expressions that the **ADEFERRED** and **TDEFERRED** commands accept are limited compared to those accepted by the other commands. This is because the targets of the deferred breakpoints do not yet reside in storage; therefore, their address expressions cannot actually reference storage. Instead, they may reference only offsets and csect names within the load modules to which they are targeted. See **HELP COMMANDS ADEFERRED** for more information.

There are no specific restrictions on the address expressions usable by the remaining trapping commands **(AT/ATX/TRAP).** See **HELP ADDRESSING** for a full discussion of address expressions.

Trapping Commands and the CDP

Like address expressions for any command, the address expressions for the **AT/ATX/TRAP** commands may make use of Z/XDC's **Current Display Pointer** to reduce the amount of typing necessary. See **HELP ADDRESSING IMPLICIT** for details about the **CDP.**

The **CDP** is particularly useful when the trapping command is given **multiple** address expressions. In this case, those address expressions that **do not** reference the CDP, **temporarily** change the CDP, while those address expressions that **do reference** the CDP do not change it. (Weird, huh?) Well, these rules have the following useful consequences:

- Commands that execute following an AT/ATX/TRAP command will not see any change in the CDP temporarily caused by the AT/ATX/TRAP command.
- A sequence of consecutive address expressions (within the AT/ATX/TRAP command) that all have implied base terms will all use the same CDP value.
- An address expression that has an explicit base term can be used to temporarily set the CDP for use by subsequent address expressions (if any) occurring within the same command.

Here's an Example:

```
FORMAT .MYSUB
TRAP   R14? +4 +8 +C .EXITVECT +4 +8 +C +10
FORMAT +0
```

- The first **FORMAT** command displays the code at **.MYSUB** and it leaves the CDP pointing there.

- The second **FORMAT** command (i.e. the third command) uses the CDP to redisplay **.MYSUB.**

- Meanwhile, the **TRAP** command (the second command) sets **nine** one-shot traps as follows:

  - It sets a trap at **R14?** and **temporarily sets the CDP** to that location.
  - Then it **uses CDP references** to set three additional traps into a return vector pointed to by R14.
  - Then it sets another trap at **.EXITVECT;** and again, it **temporarily sets the CDP** to that location.
  - Then it **uses CDP references** to set four additional traps into the exit vector labeled by .EXITVECT.
  - Then it **restores the CDP** to what it had been set to before the TRAP command started (i.e. to what it was set to by the first FORMAT command) so that it can be used by subsequent commands (such as the second FORMAT command).

- [Note, because these nine traps were all set by a **single** command, when execution reaches any one of them, they will **all** be removed. See **HELP BREAKPOINTS FAMILIES** for more information.]

The Characteristics of Traps

Traps can have any or all of the following characteristics:

- They can be **transient** or **persistent.**
- They can be **active** or **deferred.**
- They can be **conditional** or **unconditional.**
- They can have **automatic commands** associated with them.
- They can be gathered together into **families** that can be manipulated as a group.

See the **crosslinks** for the details on each of these.

Help Breakpoints TRACing

Tracing is how Z/XDC refers to process of **stepping** execution through an Assembler program. In detail, **tracing** is the automatic process of:

- Setting a breakpoint into the user program's execution path,
- Returning execution control to the user program,
- Waiting for execution to reach the breakpoint,
- Receiving control back from the user program,
- And clearing the breakpoint.

In other words, in Z/XDC **tracing** means **to step through** the execution of a program one (or a few) instruction(s) at a time:

- For **Assembler** programs, tracing is performed using the **TRACE** command.
- For **C** programs, tracing is performed using the **STEP** command.

In XL C/C++ and Metal C programs, **TRACE** commands and **STEP** commands can be freely intermixed (assuming of course that the c/XDC Feature has been licensed for use). The **STEP** command is program statement oriented, while the **TRACE** command remains machine instruction oriented.

Tracing Types

The following kinds of traces can be performed upon machine code:

Instruction Trace (T or **T I)**
Control is returned to the user program so that it can execute the machine instruction that is currently pointed to by the retry level PSW. Z/XDC receives control again prior to the **next machine instruction** to be executed thereafter. See **HELP COMMANDS TRACE I** for more information.

Branch Trace (T B)
Control is returned to the user program. Z/XDC receives control again prior to the execution of the **next branching instruction** or the next EX/EXRL of a branch type instruction. See **HELP COMMANDS TRACE B** for more information.

Successful Branch Trace (T BY)
This is a branch trace that stops only if the **branch actually will occur.** If the branch will fall through, Z/XDC will silently allow the user program to resume until it reaches the next branch. This cycle repeats until a branch is reached that will not fall thru. See **HELP COMMANDS TRACE BY** for more information.

Trace Branch-No-Call (T BNC or just **T BN)**
This is a trace that is like **T BY** except that **call-with-return** linkage instructions (BAL BAS JAS etc.) are **not** followed.

Instead, recapture traps are placed at the called subroutine’s return points, and the subroutine is allowed to run.

Effort is made to distinguish linkage instructions used for **call-with-return** purposes from those used for **point-and-jump-over** purposes:

- For **point-and-jump-over** linkage instructions, and for all other branching instructions, **T BNC** follows the jump (just like **T BY** does).

- For **call-with-return** linkage instructions, **T BNC** places recapture traps at the

called subroutine's return points, and then lets the subroutine run without trying to follow it.

See **HELP COMMANDS TRACE BNC** for more information.

Storage Alteration Trace (T SB and **T SA)**
This is an instruction trace that stops when a **storing instruction** is reached.
- **T SB** stops **before** the store type instruction is executed.
- **T SA** stops **after** executing the store type instruction.

See **HELP COMMANDS TRACE SB/SA** for more information.

Loop Trace (T ∗)

This trace allows the instruction currently pointed to by the retry level PSW to be **executed one time.** Z/XDC intercepts if the user program loops back and attempts to execute that same instruction a **second time.**

Permanently stuck branches (such as **J ∗**) are recognized and cause the trace to abort.

On the other hand, temporarily stuck branches (such as **JCT R5,∗**) are also recognized and simulated.

See **HELP COMMANDS TRACE STAR** for more information.

Trace WATCHs (T W conditionalexpression)

This is a pseudo trace that does not actually set any breakpoints. Instead, it just waits patiently until a specified condition is met.

In other words, it allows you to define one of more conditional expressions that are not associated with any particular breakpoint. Instead, whenever Z/XDC receives control for any reason (such as for other breakpoints), all conditional expressions are evaluated, and if all resolve to **FALSE,** the user program is silently allowed to resume.

For more information, see **HELP COMMANDS TRACE W.**

Conditional Expressions: The FALSE Constant

Conditional expressions can be assigned to:
- Any trap (AT, TRAP or ATX),
- Any deferred trap (AD or TD [but not HD]),
- Any trace (T whatever)
- And any WATCH (T W).

Just enclose the expression with parentheses and stick it onto whichever command you're using. For further details, see **HELP COMMANDS SYNTAX BREAKPOINTS CONDITIONS.**

Conceptually, it may be easier for you define all your conditional expressions

as **WATCHs,** and not also have to think much about the conditional expression you want to assign to your individual traps (or trace). This is why Z/XDC supports something called the **FALSE** constant for creating things called **FALSE Traps.** These are traps (or traces) with a very simple conditional expression that will **always** resolve as **FALSE.** Examples:

- **T address [address][...] (FALSE)**
  This sets one or more **FALSE Traps** at the locations you choose. (Presumedly, you'd chose locations that are useful to your current debugging scenario.)

- **T BNC (FALSE)**
  This causes Z/XDC to receive control at each branch-capable instruction that the current main routine or subroutine encounters. This gives Z/XDC the opportunity to check all enabled **WATCHs** to see whether it should:
  - Return to your debugging session (if **any** WATCH's expression resolves **TRUE),**
  - Or silently allow your program's execution to resume (if **all** WATCH expressions resolve **FALSE).**

In other words, **FALSE Traps** are an easy method of creating **WATCH** evaluation points in your code. At each point that Z/XDC receives control, if **WATCHs** exist, they will help decide whether your program will be silently resumed or noisily interrupted. See **HELP BREAKPOINTS CONDITIONAL WATCH** for more information.

The **T BNC (FALSE)** itself will not participate in the decision to interrupt execution vs. to silently resume it.

All Together Now...

The various kinds of traces can be freely intermixed.

Breakpoints used for **tracing** can have any or all of the following characteristics:

- They can be conditional or unconditional.
  (See **HELP BREAKPOINTS CONDITIONAL)**

- They can have automatic commands associated with them.
  (See **HELP BREAKPOINTS AUTOCMDS)**

- They can only be transient (never persistent).
  (See **HELP BREAKPOINTS TRANSIENT)**

- They can only be active (never deferred).
  (See **HELP BREAKPOINTS DEFERRED)**

Running the Movie Backwards

During your debugging session, Z/XDC keeps a history of your commands and displays for all activity that occurred in your display screen's **Primary Window** (i.e. your screen's top command line). This history is called the **Scroll Area.**

So every time you use a **TRACE** command, for example, the display that results when Z/XDC receives control back from your program is recorded in the Scroll Area.

You can scroll up and down through this history with simple **UP** and **DOWN** commands.

One form of the **UP** command (and **DOWN** command too) relates to tracing. The **UP TRACE;RETRIEVE** command string allows you to run your trace **backwards** just as if it were a movie!

It does this by selectively showing only those historical displays that were generated first each time Z/XDC received control back from the user program.

In case you want to go both backwards and forwards, you can intermix your **UP T** commands with **DOWN T's** to make this happen too.

The factory default PF keys define the **F7** and **F8** keys as **"UP -"** and **"DOWN -",** respectively.

Note, workstation keyboard maps often map the **PageUp** and **PageDown** keys to mirror the **F7** and **F8** keys. This means that you can easily issue multiple **UP T** commands simply by typing **T** on the command line and pressing **PageUp** multiple times. (If you do that fast, it truly looks like a movie.)

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

BRANCHES - More information about branch tracing.
STOREALTER - More information about storage alteration tracing.
SUBROUTINES - How to trace or skip tracing a subroutine.
ALIEN - Tracing branches into system routines.
EX/EXRL - Tracing through EX/EXRL instructions.
PFKEYS - Pfkeys defined for use in tracing.
CONDITIONAL - Conditional tracing.

For detailed information about **WATCHs,** see **HELP BREAKPOINTS CONDITIONAL WATCH.**

Help Breakpoints TRACing Branches

Instruction Categories

For the purposes of **branch tracing (T B, T BY** and **T BNC),** Z/XDC divides all machine instructions into four categories:

- **Linkage instructions:** Examples are BAS, BALR, JAS, BAKR, etc.

- **Branching instructions** which Z/XDC can follow: Examples are BC, BRC, Jxx, PT, BR, JXLE, LPSW, etc.

- **Transfer instructions** which Z/XDC cannot follow: Examples are SVC, PC, MC, etc.

- **Fall-thru instructions** which do not branch. Note, this includes certain forms of branching instructions that can never branch. Examples are BALR Rn,0; BCTR Rn,0; NOP; etc.

The several **LOAD-AND-TRAP (LAT)** and **COMPARE-AND-TRAP (CRT)** type instructions are considered by Z/XDC to be fall-thru instructions. The branch traces all treat them no differently from any other fall-thru instruction.

Types of Branch Traces

Z/XDC supports three kinds of "branch" traces:

Branch Trace (T B)
Control is returned to the user program. Z/XDC recaptures execution again prior to the execution of the next branch, linkage or transfer instruction (or the next EX/EXRL thereof) regardless of whether or not that instruction will or will not branch.

Successful Branch Trace (T BY)
This is a branch trace that interrupts your program's execution only on branch instructions and linkage instructions that will, in fact, branch. All transfer instructions are also stopped at.

Branch-No-Call Trace (T BNC or just **T BN)**
This is a trace that is like **T BY** except that **call-with-return** linkage instructions (BAL, BAS, JAS, etc.) are **not** followed. Instead, they are **stepped over.** Recapture traps are set at the called subroutine's return points, and then the subroutine is allowed to run without being followed.

Note, Z/XDC makes an effort to distinguish linkage instructions used for **call-with-return** purposes from those used for **point-and-jump-over** purposes. For **point-and-jump-over** linkage instructions, and for all other branching instructions, **T BNC** follows the jump (just like **T BY** does). See **HELP COMMANDS TRACE BNC** for more information.

What the Branch Traces Do

While performing any of the above types of branch traces, Z/XDC will choose its interrupt points based upon the type of branch trace being done and depending upon what category the currently reached machine instruction falls into:

- For **linkage** instructions, all branch traces will always stop. They will differ in what they will do next:
  - **T B** and **T BY** will try to **follow** to the target,
  - While **T BNC** will perform **further analysis** before deciding whether to follow to the target or recapture the return.
    - For **point-and-jump-over** usages, T BNC will **follow** the jump.
    - For **call-with-return** usages, T BNC will set **recapture traps,** and allow the subroutine to run without following it.

- For **branch** instructions, Z/XDC **may or may not** stop depending upon the type of branch tracing being done:
  - T B will **always stop.**
  - T BY and T BNC will stop **only** if the branch will, in fact branch. If it will fall thru, then they will not stop.

- For **EX/EXRL** instructions, Z/XDC may or may not stop depending upon what their target instructions are.

- For **transfer** instructions, Z/XDC will **always stop** regardless of the type of branch tracing being done.

- For **non-branching** forms of branch instructions (NOP; BALR Rn,0; BCTR Rn,0; etc.), Z/XDC will consider them to be the same as fall-thru instructions, so Z/XDC will **never** stop at them.

- For **fall-thru** instructions, Z/XDC will **never** stop unless a subroutine return recapture trap is on it.

Linkage Instructions

The following are linkage instructions:

```
BAKR  BALR  BASR  BRAS  BAL   BAS   BASSM JAS
```

Branch Instructions

The following are branching instructions:

```
BC    BCTR  BRXH  BSM   Bwhatever   LPSW  PT
BCR   BRC   BRXLE BXH   JCT   JXLE  PR
BCT   BRCT  BSA   BXLE  JXH   Jwhatever
```

Transfer Instructions

The following are transfer instructions:

```
LASP  BSG   SVC   PC    RP    MC    TRAP2 SIE
SVCST DIAG  SSM   STOSM STNSM LCTL  TRAP4
```

Five of these (SSM, STOSM, STNSM, LCTL and LASP) are not really transfer instructions, but they are **state changing** instructions that can affect the debugging environment, so Z/XDC always stops when it encounters one during a branch trace.

Fall-Thru Instruction

Fall-thru instructions are everything else. They are never stopped at by any form of branch trace unless a subroutine return recapture trap has been placed on them.

Special Cases

There also are several perhaps unexpected cases that are best handled as if they were fall-thru instructions. They are **NOPs, CRT** and **LAT** type instructions and **#DIE** macros...

Special Case - NOPs

The following **non-branching** forms of the branch instructions are treated by Z/XDC as being **fall-thru** instructions: They are **not** stopped at during branch tracing unless they are targets of the **EX/EXRL** instruction.

```
NOPR ---  BC   0,---  BALR Rn,0  BSM   Rn,0  BCTR Rn,0
NOP  ---  BCR  0,Rn   BASR Rn,0  BASSM Rn,0
JNOP ---  BCR  m,0    BAKR Rn,0
          BRC  0,---
```

Special Case - CRTs and LATs and Friends

There currently are 15 variations of the **CRT** and **LAT** machine instructions that intentionally cause a data exception when a defined condition is met. the data exception leads to an ABEND **(s0C7-FF)** which, of course, causes Z/XDC to receive control. When the exceptional condition does not occur, program execution simply continues.

These CRT, LAT and related instructions are treated by Z/XDC as being **fall-thru** instructions: They are NOT stopped at by any form of branch tracing unless the trap condition is met, in which case Z/XDC will receive control due to the resulting **s0C7-FF** ABEND.

When an **s0C7-FF** does occur (i.e. when a CRT or LAT trap is triggered), Z/XDC will receive control and will issue a special version of its **DBC851I** message that tells you what happened and what you can do about it.

- First, of course, you may want to use **WHERE, FORMAT, SHOW** and other commands to look around and figure out happened.

- Then if you want to just continue running your program as if nothing happened, then use the **GO** or **TRACE** command to resume your program with the next instruction after the CRT or LAT or similar instruction. (Your compare-and-trap exception handler, if any, will not receive control.)

- On the other hand, if your recovery routine (ESTAE[X] or whatever) has logic in it to deal with **s0C7-FF** ABENDs, then in order to pass control to it, you need to **percolate** the ABEND. You do that by issuing an **END KEEP** command. (See **HELP COMMANDS END** for details.)

  Note that the **END** command does not end your debugging session. It only percolates the current ABEND for processing by the next older recovery routine (your ESTAE[X] routine, presumedly). If your routine should recover the ABEND, then the debugging session will continue when the next ABEND occurs or the next breakpoint is reached.

Special Case - #DIE Macros

There is a special circumstance under which the **T BY** and **T BNC** traces will **not stop** at a branching instruction that **will branch...**

The **#DIE macro** is frequently used to generate conditional **DEAD traps.** These are runtime sanity checks that are preceded by a conditional branch or jump instruction that will:

- Either fall through (to the DEAD trap) if the insane condition occurs
- Or skip around otherwise.

See **HELP BREAKPOINTS DEADTRAPS #DIE** for more information.

When tracing through the execution of a program via the **T BY** and **T BNC** commands, it generally is not particularly useful for the trace to stop at these trap-bypassing branch instructions. Accordingly, z/XDC has support for a profiled setting controlling whether the **T BY** and **T BNC** traces will or will not stop. That setting is **SET TRACE STOP/IGNORE.**

This setting can be displayed and changed by the **"FORMAT TRACE and FIND Settings"** panel of the Profile Menuing System. See **HELP PROFILES MENU FORMATTRACEFIND** for more information.

This setting can also be changed by the **STOP** and **IGNORE** operands of the **SET TRACE** command. For more information, see **HELP COMMANDS SET TRACE IGNORE.**

The **LIST TRACE** command can be used to display this setting’s current value.

The factory default setting is for **T BY** and **T BNC** commands to **IGNORE** bypassed **DEAD traps.**

Help Breakpoints TRACing STorealter

What Is a Storage Alteration Trace?

Storage alteration tracing is the process of:

- Scanning forward through the instruction stream to find the next machine instruction that has the **potential** for altering anything in storage,
- Placing a temporary trap either at **(T SB)** or after **(T SA)** that instruction,
- And then allowing user program execution to run until that trap is reached.
- The trap is removed.
- Then displays are emitted, and control is given to the user.

Note, stopping will occur regardless of whether or not the instruction actually changes storage.

If a store type is the target of an EX or EXRL, that is recognized and handled correctly.

If branching instructions occur along the way, they are silently followed until a store type instruction is reached.

How To Do a Storage Alteration Trace

Storage alteration tracing is performed by repeatedly issuing two variations of Z/XDC's **TRACE** command:

- Use **T SB** to stop **at** the next store type instruction in the execution stream (i.e. **before** the store operation actually occurs).
- Use **T SA** to stop **after** the next store type instruction in the execution stream (i.e. **after** the store operation actually occurs).

See **HELP COMMANDS TRACE SB/SA** for more information.

Under the Covers

Silently, Z/XDC scans ahead in your code looking for:

- Store type instructions,
- Execution flow altering instructions (branching instructions, jumping instructions, SVCs, PCs, LPSWs, etc. etc.),
- And EXs and EXRLs.

Z/XDC then sets an interim trap on whatever it finds and then allows program execution resume.

When execution reaches the trap, Z/XDC receives control and does the following:

- It removes the trap.
- If the instruction alters the flow of execution, it sees whether or not a branch will actually occur.
- It then places a new interim trap on the next branching (or storing) instruction following either the fall-thru or the current branching instruction's jump target.
- It then allows program execution to resume.
- All this is done silently.
- The intention is to guarantee that Z/XDC does not lose control of user program execution.

On the other hand, if the reached instruction is a store type, the objective is achieved, so execution is halted either at or after that instruction, and a display is emitted, and control is given to the user.

Recapture Points

As may now be clear, for each issuing of a **T SB/SA** command, Z/XDC may receive control silently many times before a store type instruction is finally reached. Each of these interim stops is referred to as a **recapture point.** These points give Z/XDC opportunities to update information and evaluate **WATCHs** and other Conditional Expressions.

Conditional Expressions

If:

- The **T SB/SA** command is given with a conditional expression,
- Or one or more **WATCHs** exist,

Then all said conditionals are evaluated at every recapture point, and the decision is made to continue tracing or to stop, according to whether any expression evaluates **TRUE** or all expressions evaluate **FALSE.** For details, see **HELP BREAKPOINTS CONDITIONAL.**

What Are Store Type Instructions?

For the purposes of store tracing, a "store type instruction" is any instruction that has the **potential** of altering **virtual** storage.

Certain privileged instructions that are defined to use REAL addresses (regardless of DAT) are ignored for store tracing purposes.

On the other hand, store type instructions that happen to address real storage simply because DAT is off are still stopped at.

Also ignored (of course) are all instructions that are not capable of storing into virtual storage (LR for example).

What's **not** ignored (of course) are store type instructions that **explicitly** reference storage.

Also not ignored are certain instructions that **implicitly** alter storage (or are likely to). These include PLO, SVC, PC and the linkage stack affecting instructions (BAKR, PC and PT).

Stepping Over Subroutines

The **T SB** and **T SA** traces (like **T BNC)** do not trace through subroutines. They always step over them.

If a **T SB/SA** has a conditional expression (and/or if **WATCHs** exist), then those will be evaluated upon return from a subroutine, so if the subroutine makes a WATCH'd for storage change, that will be revealed.

For further details, see **HELP BREAKPOINTS TRACING SUBROUTINES.**

Best Practice

Generally, storage alteration tracing is useful only when it is accompanied by a conditional expression that is designed to filter for a particular alteration that you are trying to trap for. Without such an expression, user program execution is stopped at or after every single store type instruction encountered.

Perhaps the easiest way to do this is:

- Use one or more **T W (conditionalexpression)** commands to set up the desired conditions.

- Then issue **T SA (FALSE)** to start the trace.

Notice, as described above, the **T SA (FALSE)** will not follow execution into subroutines. But the trace will stop should the subroutine make the storage change being searched for.

See **HELP BREAKPOINTS CONDITIONAL** for more information.

Help Breakpoints TRACing SUbroutines

Stepping Over Subroutines

From time to time, while tracing through a section of code, you may come upon a linkage instruction (BAL, BASR, JAS, etc.) that calls a subroutine. If you wish to trace through the subroutine (i.e. **step into** the subroutine), then just continue issuing whichever tracing commands you are already using **(T, T B** or **T BY).**

Often, however, you may not be interested in tracing through such a subroutine, and you would prefer to just **step over** it (i.e. let the subroutine execute and not resume tracing until it returns). The best method to do this is:
- Start by determining all locations to which a subroutine might return.
- Then set breakpoints on each of those return locations,
- And then type **GO** to let the subroutine proceed.

This may seem overly complicated, but in practice it usually is not.

For one thing, there is a PF key **(F11)** that is real helpful in most situations for stepping over subroutines. It sets recapture traps on several instructions following the linkage instruction and then issues a **GO** command to let the subroutine run. See **HELP BREAKPOINTS TRACING PFKEYS** for additional information.

Also, there is the **T BNC** command that does pretty much the same thing that **F11** does: It sets recapture traps following the linkage instruction, and then allows the subroutine to run. But it also handles **EODAD** and **SYNAD** exit routines when they are a factor.

Vectored Returns

In most cases, a subroutine will return only to the location immediately following the instruction that calls it. But in some cases, a subroutine will be written to return to any of a series of instructions following the calling instruction. This is sometimes referred to as a **vectored return,** and the possible return points are called a **return vector.** They typically consist of several 4-byte branching instructions following the linkage instruction. Example:

```
    JAS  R14,SUBRTINE
    B    NOK             +0
    J    AOK             +4
    B    CHEKMORE        +8
```

Both PF key **F11** and the **T BNC** command are specifically designed to correctly handle return vectors ranging from one to six instructions long:

- In the case of **F11,** the vector is presumed to be six instructions long. Notes:
  - It does not matter if those instructions are a mix of 2-byte, 4-byte or 6-byte instructions of just data.
  - It also doesn't matter when this presumption is wrong. Any damage done by the placement of excess recapture traps is repaired before that damage matters.

- In the case of **T BNC,** short return vectors are recognized either when a 2-byte or 6-byte instruction or data is found. So **T BNC** may set fewer than six recapture traps.

Subroutines that Don't Make Normal Returns

The step over support provided by **F11** and **T BNC** does not cover all possible cases. Examples:
- Return vectors that are 7 or more slots long,
- Return vectors consisting of instructions whose lengths are not all 4 bytes.
- Exits or aborts that are made by the subroutine to other locations.

In these cases, you will need to know what the possibilities are, and manually use **AT** or **TRAP** commands to set recapture traps at every exit and abort target and vector slot that can occur.

When a subroutine jumps to an exit or abort target, if there is a TRAP-type trap there, it will be automatically removed. If it's an AT-type trap, it will remain.

If you have additional traps located at other exit or abort targets, typically you would like them removed also. Well, if all the traps are assigned to the same family, then all the TRAP-type traps will be removed. See **HELP BREAKPOINTS FAMILIES** for more information.

If the routine being stepped over is an **access method,** and if it is possible for it to jump to a **SYNAD** or **EODAD** routine, then:
- **T BNC** will recognize the possibility and properly handle it.
- **F11** will not.

Subroutines in Alien Storage

A perfect example of a subroutine that you **don't** want to trace through is any routine that is a part of the operating system. Most such are accessed via SVC and PC instructions. You don't need to worry about those, because no **TRACE** command will ever attempt to step into those. Z/XDC will always place its recapture trap following the SVC or PC, and then let execution run.

The common situation that does occur is when the service routine is called using a BASR (and friends). This is where you would want to use **F11** or **T BNC** to step over the routine.

But you might not need to. If:

- **SET TRACE LOCAL** is in effect,
- And Z/XDC detects that the service routine is located in **alien storage,**

then all forms of the **TRACE** command will step over (not into) the subroutine.

For more information, See **HELP BREAKPOINTS TRACING ALIEN.**

Help Breakpoints TRACing Alien

What is Alien Storage?

In order to mitigate the possibility of tracing execution into programs that are of no interest to you or that are dangerous to step through, Z/XDC has a concept of **alien storage.** Then whether or not Z/XDC will allow a trace to enter alien storage can be controlled by the **SET TRACE GLOBAL|LOCAL** setting. See **HELP COMMANDS SET TRACE LOCAL** for more information.

When **SET TRACE LOCAL** is in effect, and the user program's execution has reached a branching instruction (BR, J, BASR, etc.), Z/XDC tracing will check to see whether or not the branch target is in **alien storage.**

What is considered to be alien storage differs according to whether the user program is running authorized or non-authorized.

- When Z/XDC is running **non-authorized,** alien storage is any **protected** area of storage. This generally includes all transfers to store protected routines, routines located in common storage and all transfers to reentrant and refreshable routines.

  (Note, The System usually loads reentrant and refreshable programs from authorized libraries into key 0 storage, but Z/XDC provides ways to change that behavior. See **HELP DEBUGGING REENTRANT** and **HELP DEBUGGING REFRESHABLE** for more information.)

- When Z/XDC is running **authorized,** all of the following are considered to be transfers to alien storage:

  - Any transfer from the private area to any routine in common storage.

  - Any transfer from any common area module to any other common area module.
    Notes:
    - Any transfer within a common area module is NOT considered to be an alien transfer.
    - Note that the System Nucleus is considered to be a single module.

  - Any transfer from the common area to the private area also is **not** considered to be an alien transfer.

  - Any transfer to the Home Address Space from another address space (via the PT and PR instructions) is **not** considered to be an alien transfer.

- Any transfer to the private area of a foreign address space from the same foreign address space (via the PR and PT instructions) is **not** considered to be an alien transfer.

What Happens When Alien Storage is Encountered?

When Z/XDC is processing trace requests, it must consider whether the user program is about to transfer to alien storage. There are two circumstances:

- When Z/XDC is running **non-authorized,** then tracing execution into alien storage is simply not possible. This is because alien storage is (by definition) protected and, therefore, Z/XDC is unable to set the necessary breakpoints. Accordingly, any attempt to use the tracing commands to follow execution into alien storage will be rejected by Z/XDC.

- When Z/XDC is running **authorized,** then it is capable of tracing execution into alien storage; however, doing so is very dangerous and should not be done except with extreme caution! Accordingly, when Z/XDC detects an alien transfer during tracing, It will do one or the other of the following, depending upon the current **SET TRACE GLOBAL|LOCAL** setting:

  - If **SET TRACE GLOBAL** is in effect, then Z/XDC will stop and issue a message **(DBC803Q)** to ask the user if he really really wants to proceed, and the branch will be allowed or rejected according to his response. This reduces the likelihood of the user accidentally setting a breakpoint in such critical system routines as, for example, the dispatcher.

  - On the other hand, if **SET TRACE LOCAL** is in effect, then the branch will always be rejected. Z/XDC will simply not continue the trace into the alien storage.

Rejected?

But rejection is not the end of the world. The trace may continue anyway. It depends upon whether or not the branching instruction is a **linkage** instruction.

- If the branching instruction is a linkage instruction (BAS, BAKR, BAL, JAS, etc.), then Z/XDC will set recapture traps at the return address and then allow execution to continue into the alien subroutine. When execution returns, Z/XDC will recapture control and tracing will continue.

- But if the branching instruction is any other kind, then Z/XDC aborts the trace and issues message **DBC184E.**

Using SET TRACE LOCAL to Step Over Alien Subroutines

Under most circumstances, it is likely that the user is not really interested in tracing through a system routine and that he wants to let it execute, without interception, until it returns to its caller. (This is called **stepping over.)**

If this is true, then the user can use the **SET TRACE LOCAL** command to tell Z/XDC to

step over subroutines located in alien storage. So, Z/XDC assumes that the subroutine will return to the next one or more instructions following the linkage instruction (i.e. into a **return vector).**

Accordingly, when tracing, Z/XDC sets its recapture traps into the return vector instead of at the target subroutine. For more information about this, see **HELP COMMANDS SET TRACE LOCAL.**

SET TRACE LOCAL is especially useful for conditional traces. If **TRACE LOCAL** is not set, then every time execution reaches a branch into alien storage, the conditional trace aborts either with a **DBC184E** error message (if running non-authorized) or with the **DBC803Q** query (if running authorized) asking for permission to set the next breakpoint. For more information about conditional traces, see **HELP BREAKPOINTS TRACING CONDITIONAL.**

The **SET TRACE LOCAL** command prevents conditional traces from being prematurely aborted, but on the other hand it creates a possibility that the conditional trace will lose control of execution should the subroutine exit to a different location then the return vector. (See below for special recognition and handling of SYNAD and EODAD exits.)

Using T BNC to Step Over All Subroutines

When a **Branch-No-Call** trace is used, alien storage checks are not needed.

There are several types of traces (see **HELP COMMANDS TRACE).** One of them **(T BNC)** is specifically designed to **step over** all subroutines, whether in alien storage or not.

T BNC is the **Branch-No-Call** trace. It behaves exactly like **T BY** except when the current machine instruction is a **linkage instruction** that is about to call a subroutine...

- In the **T BY** case, an alien storage check is made, and the trace will either step over or step into the subroutine according to whether or not the subroutine is located in alien storage:
  - When **T BY** steps **over** a subroutine, it sets recapture traps at the return points and allows the subroutine to run without being followed.
  - When **T BY** steps **into** a subroutine, it sets its next trap into the subroutine so that the trace may follow execution into it.

- In the **T BNC** case, an alien storage check is not done, because if the linkage instruction is being used to call a subroutine, it is always not followed. Instead, recapture traps are set at the return points, and the routine is allowed to run without being followed. In other words, **T BNC** will always step over all subroutines regardless of where they are located.

Effort is made to distinguish linkage instructions used for **call-with-return** purposes from those used for **point-and-jump-over** purposes:

- For **point-and-jump-over** linkage instructions (as well as for all other branching instructions), **T BNC** follows the jump (just like **T BY** does).

- For **call-with-return** linkage instructions, **T BNC** tries to step over it. It places recapture traps at the called subroutine’s return points, and then lets the subroutine run without trying to follow it.

See **HELP COMMANDS TRACE BNC** for more information.

Access Methods and EODAD and SYNAD Exits

Access methods are the z/OS routines that are called to service GET, PUT, READ, WRITE and CHECK requests. They require special handling because, while usually they will return to the code that called them, they also can instead **exit** to EODAD and SYNAD routines to handle end of data and I/O error conditions, respectively.

Z/XDC has special logic to recognize access methods, so that when **T BY** and **T BNC** setup to step over them, additional recapture traps are set into the EODAD and SYNAD routines if they are present. (Z/XDC examines DCBs, DECBs and ACBs, as necessary, to find the exit routines.)

Help Breakpoints TRACing Ex/exrl

Z/XDC has nearly complete support for handling breakpoints placed either on an **EX/EXRL** instruction or on its **target** instruction or on **both!** In general, it doesn't matter what the target instruction is or where it is. All forms of tracing will process correctly. All but one, that is.

The One Scenario That DOESN'T Work:

If your program code contains an EX or EXRL instruction that is **immediately followed** by its target instruction, then Z/XDC may not be able to trace through that sequence. The problem is this. Depending upon the type of trace requested and the type of instruction being EX'd, it is possible for an interim trap to be always set at the **EX/EXRL's** target, and this persistence will prevent the program from ever progressing.

If your **EX/EXRL** targets are placed anywhere **except** immediately following the **EX/EXRL** instruction, then Z/XDC will always handling tracing through the **EX/EXRL** correctly.

In particular, if you like to code your **EX/EXRL** targets in line and near the **EX/EXRL** itself, then code the targets immediately **before** (not after) the **EX/EXRL** instruction. That is always a Z/XDC safe thing to do.

Examples:

```
...
EX    R5,*+4
LA    *-*,IHADCB
...
```

In this sequence, a **TRACE** commands will get hung up at the **EX/EXRL** instruction, but a **TRACE BY** or **TRACE SB** command (for example) will not.

```
...
EXRL  R5,*+6
BC    *-*,SKIPIT
...
```

In this sequence, all forms of tracing will **always** get hung up at the **EX.** This is true regardless of the kind of **TRACE** command issued.

```
...
JNOP  SKIPIT
EXRL  R5,*-4
...
```

This sequence is **always** Z/XDC safe. No matter what kind of tracing is being done, no matter what the **EX/EXRL** target instructions are, tracing will behave correctly.

Help Breakpoints TRACing Pfkeys

Z/XDC's factory default PF keys for Assembler Programs include three definitions that are particularly useful for tracing. (For complete information about Z/XDC's default PF key settings, see **HELP FULLSCREEN PFKEYS DEFAULTKEYS SET-A.)** These definitions are:

- **F9: T**
- **F10: T BY**
- **F11: T NXSEQ() NXSEQ(2) NXSEQ(3) NXSEQ(4) NXSEQ(5) NXSEQ(6);GOT**

The above PF key values are the Factory Defaults for **Assembler** programming. When using **c/XDC** to debug XL C/C++ and Metal C programs, the Factory Defaults for these three PF keys change to various **STEP** commands. See **HELP BREAKPOINTS STEPPING-C C PFKEYS** for more information.

- To load a factory default profile designed for Assembler programmers, issue **PROFILE RESET ASM.**

- To load a factory default profile designed for C programmers, issue **PROFILE RESET C.**

The Tracing PF keys Explained

For Assembler programming, I have found the above three PF key settings to be most useful when stepping through code. I highly recommend that you take the time to learn them and use them in your own development work.

The first two **(T** and **T BY)** are pretty straight forward. **F9's T** command is short for **TRACE I,** and it's for stepping through your code one instruction at a time. Each

time you send the **T** command, the current instruction is executed, and execution is then halted with the next instruction that is now current.

F10's T BY means "Trace only successful branches". Each time you press the key, execution proceeds to the next branch-type instruction that will, in fact, branch. When you press **F10** again, the current branch is taken, and execution proceeds to the next following branch that will occur.

(Fall-thrus appear to be not stopped at. But actually under the covers, Z/XDC receives control for **all** branching instructions and determines at its execution time whether said instruction will branch or fall through. If the latter, then Z/XDC advances the trap point to the sequentially next "branching" instruction and then silently allows your program to resume. For further details, see **HELP BREAKPOINTS TRACING BRANCHES.)**

F10 is an extremely useful PF key. **T BY** is a very efficient way in which to desk check new code. With **T BY** and a yellow hilighter, you can progress rapidly down your listing, seeing exactly what does and does not get executed, and why.

F11 is a bit more complicated. Its definition is:

```
T NXSEQ() NXSEQ(2) NXSEQ(3) NXSEQ(4) NXSEQ(5) NXSEQ(6);GOT
```

In plain English, this means:

- Place temporary breakpoints onto each of the six machine instructions **following** the current instruction. **(NXSEQ(n))**
- Then let the current instruction execute. **(GOT)**
- Then when execution returns to any of the six following instructions, execution is interrupted,
- And all six of the temporary breakpoints are removed.
- Then the program code is displayed and your keyboard is unlocked.

HUH?!?!???

Well, there are some common coding scenarios where PF key **F11** is **extremely useful...**

Stepping Over Subroutines Using F11

Suppose you're tracing along with **T's** and **T BY's** and then execution reaches a **linkage** instruction (BASR, BAS, BALR, BAL, BASSM, JAS, etc.) that is about to pass control to a subroutine. Now, maybe you'd want to continue tracing execution into that subroutine **(step into),** but then on the other hand, maybe you don't **(step over).**

Well, if you do want to **step into** the subroutine, then just continue doing the **T's** and **T BY's.**

But if you don't, if you just want to let that subroutine execute without tracing it (i.e. **step over** it), then simply press **F11.** It will set breakpoints onto the six instructions following the BASR (or whatever), and then it will let the BASR execute. Execution will then proceed into the subroutine, but tracing won't. Instead, control of execution will be recaptured when the subroutine returns (assuming of course that it **doesn't exit** to somewhere else).

Why six instructions? Often, subroutines will be written to make "vectored returns",

i.e. instead of returning just to the instruction immediately following the BASR, it might return to the second instruction following the BASR, or the third, or the fourth, ... So **F11** sets six traps in order to recapture execution regardless of the return (at least in most, typical situations).

The fact that most subroutine calls do not have a six slot return vector following them **does not matter** in the slightest! **F11** goes ahead and sets its six recapture traps anyway. But that doesn't matter because when return does occur (usually to the 1st instruction following the call), all six traps are immediately removed, restoring original opcodes (or data) to where they belong. So by the time execution reaches those instructions, they are fine.

So, when **T'ing** or **T BY'ing** through your code, if you come to a BASR to a subroutine that you don't want to trace through, just press **F11.** When execution returns from that subroutine, you can resume using **T's** and **T BY's** to continue your tracing.

Warning: If a subroutine happens to make a vectored return to the **seventh** instruction following the BASR, or if it simply **exits** to somewhere else altogether, then **F11** will not work. It would fail to recapture program execution since the recapture traps that it sets would be missed. So be careful of this possibility.

Stepping Over Subroutines Using T BNC

Another way to step over a subroutine is to use the **T BNC** command. This works pretty much the same way as **F11** except that it has additional logic to recognize when the subroutine being called is an **access method** (GET, PUT, READ WRITE, CHECK, etc.), in which case it will place additional recapture traps at any **SYNAD** and **EODAD** exit routines that might exist. For more information, see **HELP COMMANDS TRACE BNC.**

Calling Subroutines in Alien Storage

If you've arrived at a linkage instruction that is about to branch to a system routine (GET, PUT, READ, WRITE, etc.), chances are that such a branch would fall within Z/XDC's definition of "branching to alien storage". (See **HELP BREAKPOINTS TRACING ALIEN** for more information.) If that's the case, and if **SET TRACE LOCAL** is in effect, then two things happen:

- First, using **F11** would not be necessary. Just continue doing **T's** and **T BY's,** and the subroutine will be bypassed automatically.

- Second, if Z/XDC recognizes the alien routine as being a common access method routine, Z/XDC will examine the associated DCB, DECB or ACB and place recapture traps on any associated **EODAD** and **SYNAD** exits that it sees. So in this special case, control of execution will not be lost even when end of file and/or I/O error conditions occur.

- Warning! If you do use **F11** in this scenario (instead of switching to **F9, F10** or **T BNC),** then the just described **EODAD** and **SYNAD** recognition does not occur.

Dropping Through Loops

If execution has reached the **bottom** of a loop (BCT, BCTR, JCT, BXLE, etc.), and if you then want to let the loop continue executing to completion without tracing it any further, it's pretty easy to set up recapture traps that let the loop run to completion and recapture execution either when it jumps out of the loop or falls through the bottom.

- First, run through the loop one (or more?) times.

- When you've had enough, bring the loop to its bottom instruction (usually a Jxx, Bxx, BR, BCT, JXLE, whatever).

- Then use the **FORMAT** command to display the loop's code.

- Identify all the branching or jumping instructions by which the loop might exit prior to completion.

- Use **T** point-and-shoot commands to place recapture traps at the **targets** of those branches or jumps. (You do this by placing **T's** onto the s-cons of branching instructions and onto the RI fields of jumping instructions. See **HELP PASCMDS INSTRUCTIONS T-EXAMPLE.)**

- Press **F11.** (Yeah. I said F11.)

F11 will place additional recapture traps onto six instructions following the bottom of the loop, and the BCT (or whatever) at the bottom of the loop will be allowed to execute. Z/XDC will recapture control either when execution jumps out to one of the recapture traps set by your **T** point-and-shoots, or when it falls through the bottom of the loop.

Note, when **F11** is used, **T** point-and-shoots do not have to be used for those branching/jumping instructions that target to within six instructions of the bottom of the loop. (They can be used, but they don't have have to be used.)

Special Considerations

The above described settings for PF keys **F9, F10** and **F11** are well suited for **Assembler** debugging; however, they are **poorly suited** for **C** debugging. Therefore, when Z/XDC detects that the program, being debugged was written in XL C/C++ or Metal C, the factory default values for these PF keys are changed to **STEP** commands (not TRACE commands). For more information, see:

HELP BREAKPOINTS STEPPING-C C

HELP COMMANDS PROFILE RESET

As you can no doubt figure out, the three tracing/stepping PF key definitions supplant the normal settings of **SWAP, RIGHT** and **LEFT** for PF keys **F9, F10** and **F11. SWAP, RIGHT** and **LEFT** are, however, still useful in Z/XDC, so these definitions are still available via **shift-F9, shift-F10** and **shift-F11.**

Help Breakpoints TRACing Conditional

Conditional expressions can be used on **any** kind of breakpointing command. This includes any kind of **TRACE** command. Thus, Z/XDC supports not only conditional traps, but also **conditional traces!**

How Does a Conditional Trace Work?

A conditional trace is a trace that follows user program execution until the specified condition yields a **TRUE** result.

When you start a trace (**T BY, T BNC, T SB,** etc.), Z/XDC scans forward in your code to find the next instruction where it should set a **recapture trap.** (The particular recapture point chosen depends upon the type of trace being conducted.) Then Z/XDC allows the program to resume execution.

When execution reaches the recapture trap, Z/XDC first checks all **WATCHs** to see if any resolve **TRUE.** If so, then the trace is interrupted, and the debugging session is resumed.

If not, i.e. if all **WATCHs** resolve **FALSE,** then Z/XDC next determines if the **requirements of the trace** are met (For example, "that the branch will occur" in the case of T BY traces.) If not, then the code is scanned for the next trace point, and the cycle repeats.

If the trace **requirements are met,** then Z/XDC checks the **trace's own conditional** (if any). If that resolves **FALSE,** then the code is scanned for the next trace point, and **the cycle repeats.**

If the **trace's own conditional** resolves **TRUE** (or if there is no conditional), then the trace is interrupted, and the debugging session is resumed.

The Difference Between WATCHs and a Trace's Own Conditional

Notice that all **WATCHs** are checked at **all** recapture points, regardless of whether or not a trace's particular requirements are yet met.

On the other hand, a **trace's own conditional** is checked only after the trace's particular requirements **are** met.

This little factoid may affect your decision of where to place a conditional expression, on a **T W** command or on the **TRACE** command.

What Happens at Recapture Points

When stepping a trace through code, Z/XDC has to set **intermediate recapture traps** where it can check to see whether or not the **requirements of the trace** are yet met.

For example, for a **T BY** trace, a recapture trap needs to be placed on the next

following branching instruction. When program execution reaches that instruction, only then can an accurate determination be made as to whether or not that instruction will actually branch. In the **T BY** case, if the branch will not occur, then the trace requirement is not yet met.

When a trace's requirements are not yet met, a next recapture point has to be found, and program execution must be allowed to proceed to that point.

In order for this process to work:

- Z/XDC's analysis of whether or not a branch will occur has to be perfectly accurate. (It is.)

- Z/XDC's scan for the next recapture point also has to be perfectly accurate. (It is.)

  (Well, it is perfectly accurate except when a **T BNC** trace reaches a linkage instruction that calls a subroutine to be stepped over. Then assumptions have to be made that might not be correct. For details, see **HELP COMMANDS TRACE BNC.)**

Example:

T W (R7 EQ 00000000) 'ALARM! R7 has been clobbered!'
T BNC (.FLAGS,AND,C0,GE,40)
I'm trying to do two things here:
- Detect when either bit **X'80'** or **X'40'** is turned on in a field named **FLAGS.**
- Detect when **R7** gets clobbered.

So the **T W** is set up to detect the R7 problem, and the **T BNC** command is issued to detect the flag settings that I care about. Here's what happens upon the **T BNC** command being issued:

- The user code is scanned, and a recapture trap is set at the next branching instruction (or PC or SVC or etc.) or **EX/EXRL** thereof.

- The user program is then allowed to run to that recapture point.

- Each time Z/XDC receives control, it evaluates whether to accept the trap and resume the debugging session, or to find the next recapture point and silently resume program execution.

- For most branching instructions, the next recapture point will be the branching instruction following the target of the current branching instruction (assuming, of course, that the current branching instruction actually branches).

- But if the current instruction is a linkage instruction, then for **T BNC, T SB** and **T SA** traces, if analysis determines that the instruction is being used for calling a subroutine, then the next recapture point will be a presumed return point from that subroutine. (See **HELP COMMANDS TRACE BNC** for more information.)

- The return address prediction is not perfect, so certain tracing failures may result. See **HELP BREAKPOINTS TRACING CONDITIONAL FAILURES** for more information.

In the current example, the following happens:

- Every time Z/XDC receives control, the **T W's** conditional is checked **regardless** of whether or not the branch at the current recapture point will be taken.

- The **T BNC's** conditional is checked only at branching instructions that will, in fact, branch (i.e. at recapture points where the tracing requirements **are met).**

- In either case, if the trace requirements have **not yet** been met, and if all tested conditionals resolve **FALSE,** a new recapture point is chosen, and the user program is silently resumed.

- On the other hand:
  - If any **WATCH** resolves **TRUE,**
  - Or if the tracing requirements **are** met **and** the trace's own conditional resolves **TRUE,**

  Then the trace is interrupted, and your debugging session is resumed.

Where Recapture Points are Placed

Just where the next recapture trap is placed is determined by the kind of trace being conducted:

- **Instruction Trace (T** or **T I)**
  The next recapture point is the next instruction to be executed.

- **Branch Trace (T B** and **T BY)**
  The next recapture point is the next branch type instruction (or SVC or PC, etc.) to be executed (or EX/EXRL thereof).

- **Branch-No-Call Trace (T BNC)**
  This is like a **T BY** trace except when a **linkage** instruction is reached. Then an analysis is done to see whether the linkage instruction's usage is for **point-and-jump-over** or for **call-with-return:**

  - If the former, then the jump is followed, and the next recapture trap is placed on the next branching instruction at or past the jump target.

  - If the latter, then a set of recapture traps are placed at the subroutine's presumed return points. (Warning! it is possible for Z/XDC's assumptions to be incorrect!)

  For all the gory details, see **HELP COMMANDS TRACING BNC.**

- **Storage Alteration Trace (T SA** and **T SB)**
  These are like **T BNC** traces except that in addition to all the T BY and T BNC recapture points, additional recapture points are placed **either on or**

after store type instructions:

- For **store-after** tracing, the next recapture point will be immediately after the next store type instruction.

  (Note, for the purposes of T SA tracing, PC and SVC instructions are considered to be store type instructions, so the trace stops on the instruction that follows the SVC/PC instruction.)

- For **store-before** tracing, the next recapture point is simply the next store type instruction itself (also SVC/PC instructions).

- **Loop Trace (T ∗)**
  This is like a trap set by the **AT** command. The next recapture point is when execution returns to the current instruction.

Recommendations

If you want to do a conditional trace, we strongly recommend that you **do not use** T, T I, T B or T BY for the trace! This is because those traces **do not stay within the local code.** Instead, they will **step into subroutines,** and that's a rabbit hole that Z/XDC likely will not be able to climb out of! For more information, see **HELP BREAKPOINTS TRACING CONDITIONAL FAILURES.**

Instead, you can use conditional **T BNC, T SB** and **T SA** traces. They will attempt to **step over** subroutine calls, so the rabbit hole is avoided.

One downside of Conditional traces is that they can be **time consuming.** Often, strategically placed conditional traps can be just as effective and a lot(!) quicker.

T BNC, T SB and **T SA** traces are vulnerable to losing control of user program execution when setting return capture traps following linkage instructions. That's because predicting the return points from subroutines is not a perfectable process. Programmers can be unpredictively creative when designing subroutine return mechanisms. See **BREAKPOINTS CONDITIONAL TRACING FAILURES** for more information.

As mentioned above, **T, T I** and **T BY** must **never** be used for conditional tracing.

Because conditional traces can lose control of execution, it generally is a good idea first to place **defensive traps** at one or more points in your program that you know will execute after the point at which the problem that you are trying to track down should have occurred.

Alternatively, you can use a **countdown WATCH** to end the trace after so many recaptures.

Example: T W (1000) stops a conditional trace after 1000 recapture traps have been reached. This is **all** recapture traps, both where trace requirements are met and unmet.

Finally, you will have to be extremely patient. Conditional tracing is several orders of magnitude slower than normal execution! Be prepared to take a coffee break, go to lunch, go home and come back the next day even.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

MULTIPLE - How to use multiple traps and WATCHs to trap for multiple conditions.
CORRUPTIONS - How to use the **T BNC** command to find random storage corruption bugs!
PERFORMANCE - What to avoid to keep from degrading conditional trace performance any further than necessary.
SLIPVSXDC - Using z/OS's SLIP facility vs. using conditional tracing.
FAILURES - Understanding when conditional traces lose control of user program execution.

Help Breakpoints TRACing Conditional Multiple

When One Address has Multiple Traps Set On It

Z/XDC permits multiple breakpoints to be assigned to the same location. So:
- If a location has two (or more) breakpoints set,
- And if those breakpoints are conditional,
- And if program execution reaches that location,

Then:
- Z/XDC receives control and evaluates **all** conditionals located at the reached location.
- Z/XDC also evaluates all enabled **WATCHs.**
- If **any** conditional or WATCH resolves **TRUE,** then user program execution is interrupted.
- Of all the conditionals that resolved TRUE, if any of them have **automatic command strings,** then all such strings are queued for processing.

In other words, placing multiple conditional breakpoints at the same address results in a logical **OR'ing** of those conditionals as well as of all enabled WATCHs.

Z/XDC permits the creation of **multiple WATCHs.** These are conditional expressions that are not associated with any particular breakpoint. Instead, all **enabled** WATCHs are evaluated every time Z/XDC is entered for any conditional trace, or conditional trap as well as any trace **recapture trap.** (See **HELP BREAKPOINTS TRACING CONDITIONAL** for information about recapture traps.)

If any WATCH evaluates **TRUE,** then user program execution is interrupted. If all

WATCHs evaluate **FALSE,** and if the conditional event for which Z/XDC was entered also evaluates **FALSE,** then user program execution is resumed until the next conditional trap or trace recapture point is reached. See **HELP BREAKPOINTS CONDITIONAL WATCH** for more information.

Setting Multiple TRAPs

Placing multiple traps at a location is pretty straight forward. Just issue two or more **TRAP, AT** or **ATX** commands, each specifying the same address but different conditional expressions and/or automatic command strings.

Example:

```
TRAP .PUTMSG (XADDR(R14?)&7FFFFFFF,EQ,056ABD42) 'F R1? DATA'
TRAP .PUTMSG (R1?,EQ,'ERR301T')
```

These two traps cause execution to be interrupted when it reaches the **PUTMSG** subroutine **and:**

- **R14** points back to location **X'056ABD42'** (i.e. a particular caller), **or**
- **R1** points to text: **C'ERR301T'.**
- In the event that the first trap resolves **TRUE,** the message pointed by **R1** is displayed.

Conditional Tracing

When using WATCHs, it's fairly easy to run a trace with multiple conditionals. To do this:

- Define one WATCH for each conditional that you want to test for.

- Then define a conditional trace with a condition that always resolves **FALSE.** (I call this a **FALSE trace.)**

Example:

```
T W (R1,EQ,00000000)    'L REGS;WHERE'
T W (R4?,NE,'I-CATCHR') 'D R4?'
T W (.DCBOFLGS,AND,01,EQ,00)
T W (1000)
```

```
T BNC (FALSE)
```

Here's what's going on...

- The first three **T W** commands define three conditional expressions as WATCHs.

- In addition, two of them define automatic commands to be executed if (and only if) their owning WATCHs are the ones that evaluate **TRUE.**

- The **T BNC (FALSE)** command starts a **FALSE trace** that, but for the WATCHs, would run forever.

- The **T W (1000)** command sets a countdown limit on how long the trace will be allowed to run.

While in this example, I have chosen to set up the T BNC trace as a **FALSE trace,** it

is both possible and reasonable to use **any conditional** you like on the **T BNC** command. Doing such just defines an additional condition for which user program execution can be interrupted.

Help Breakpoints TRACing Conditional Corruptions

What are the Hardest Bugs to Find?

Some of the hardest bugs to solve are random storage corruptions! This is where some location is being altered by unknown code but the consequences of that alteration do not manifest until millions of instructions later. By the time a failure finally occurs, the culprit will be long gone! How on Earth do you find such an assassin?

What are the Tools?

When trying to identify who is causing a storage corruption problem, there are two tools to consider: IBM's **SLIP** command and our **Z/XDC** product. Each has advantages, each has disadvantages, so which should you choose?

Well, there's a lot of nuances to the choice, so for a comprehensive discussion, see **HELP BREAKPOINTS TRACING CONDITIONAL SLIPVSXDC.** But here are the main considerations in plain terms.

- If the corruption is simple, **SLIP** is the better tool.
- If the corruption is complex, then you're probably better off using **Z/XDC.**

Simple vs. Complex Corruptions

SIMPLE CORRUPTIONS:

- The location being corrupted is static. It doesn't move around.
- The content of the location is expected to be static. It should never change.

COMPLEX CORRUPTIONS:

- The location being corrupted moves around. It changes from one incident to the next.
- The location being corrupted is one whose content is expected to change, so the corruption is not that the location is being changed. It's about what the change is.

Prerequisites for using Z/XDC

Before you can use **Z/XDC** to shoot a storage corruption bug, two things must be true:

1) You must have some idea which program is causing the corruption.

2) **Z/XDC** must be set up to debug that program. (A cross memory **HOOK** command might be a good way to make this happen.)

A Scenario

Suppose the following:

- You have a typical program that calls subroutines.
- Those subroutines call other subroutines.
- And so on to whatever depth.

- Somewhere, deep in the depths of the subroutine calls, someone is corrupting a chain field in some element of a control block chain.
- Exactly which element is changed may vary from one incident to the next.

- You have no idea who's doing that and where it's happening. You just know that when your program starts, everything's cool, but eventually someone somewhere fails because of the corruption.

Maps will be Needed

The troubleshooting process I'm about to describe relies heavily on the **TRACE BNC** command. That command performs a **step over** trace for Assembler code. It is designed to:

- Recognize **linkage instructions** that will branch to subroutines (BAL, BAS, BALR BAS, JAS and the like),

- And **not follow** that branch,

- But instead set **recapture traps** onto instructions that the subroutine is likely to return to.

There are several difficulties that the underlying **T BNC** logic has to cope with. One arises when a subroutine makes its return to someplace unexpected. See **Conditional Trace Failures - Missed Recapture Traps** in HELP BREAKPOINTS TRACING CONDITIONAL FAILURES for more information.

The other is distinguishing between linkage instructions that call subroutines vs. those that point to an inline chunk of data and at the same time jump over it. (Many z/OS macros do this!)

- In the former case, **T BNC** needs to set its traps at the likely return points.

- In the latter case, **T BNC** needs to set its trap at or following the instruction's jump-to point.

See **Linkage Instruction Usage Analysis** in HELP BREAKPOINTS TRACING CONDITIONAL FAILURES for a detailed discussion if this issue.

The point I want to make here is this. **The accuracy of T BNC's analysis is greatly improved if a map has been loaded** for the csect containing the linkage instructions that have to be stepped over by the conditional trace. For more information about

creating, loading and using maps, see **HELP MAPS.**

The Process

To find the storage corrupting culprit, try the following:

1) You will need a Z/XDC debugging session to be active in the suspect program. If you don't already have one running, you can probably use a cross memory **HOOK** command to start one.

2) Use **TRACE WATCH** commands to create one or more Watch Conditionals:

   - A Watch is a conditional expression that gets evaluated every time Z/XDC receives control.
   - Typically, when processing certain trace commands (particularly **T BY** and **T BNC),** Z/XDC receives control frequently but silently to see if the trace's criteria has been met.
   - When all Watch Conditionals evaluate **false,** user program execution is silently resumed.
   - When any Watch Conditional evaluates **true,** user program execution is interrupted, Z/XDC produces its displays, the keyboard unlocks, and you can look around to see what happened.

   See **HELP BREAKPOINTS CONDITIONAL WATCH** for more information.

3) Issue **TRACE BNC (FALSE)** to start the conditional trace. This will do the following:

   - It will cause the current routine or subroutine to start running.
   - As it runs, Z/XDC will, under the covers, receive control on every branching instruction that execution reaches.
   - (Actually, Z/XDC will receive control on **every** machine instruction that has the potential to change the PSW's next instruction address, not just vanilla branching instructions.)
   - Every time Z/XDC receives control, it evaluates all Watch Conditionals to see if any of them are now true.
   - The **T BNC (FALSE)** itself will not participate in the evaluation.
   - If all Conditionals evaluate **false,** the subroutine is silently resumed.
   - Eventually, when one or more Conditionals evaluate **true,** corruption has occurred! Subroutine execution is interrupted, and you can now look around to see what happened.

4) Execution will have stopped either on a branching instruction or on a return after a linkage instruction that has called a subroutine. (Such is the nature of a **T BNC** trace.)

5) If execution is on a branching instruction, then the corrupting instruction is one of a handful of previously executed instructions. Often, you will now be able to spot the miscreant simply be examining the nearby code. But if not...

- Use the **LIST BRANCHES** command to see a report of recently executed branching instructions.
- It is guaranteed that the corruption was perpetrated by a machine instruction located between the last branch listed in the report and the current instruction.
- If you still don't see the problem, then rerun your test, but this time, use the **AT** command to place a trap on that last instruction listed in the **LIST BRANCHES** report.
- If that instruction is executed multiple times prior to the corruption happening, you may have to make the **AT** command conditional.
- Issue a **GO** command to let execution run until it reaches that instruction at the right point in time.
- Once you arrived at the code that commits the corruption, at the point in time when the corruption is going to be committed, start using the **T** command to step through the code one instruction at a time until **BOOM** happens.

6) On the other hand, if execution stopped at a return from a subroutine call, then you now know that the corruption occurred somewhere within that subroutine.

You will have to abandon this test run and start over, only this time...

- Set up your Watch Conditionals the same as before, but don't issue the **T BNC (FALSE)** yet.
- Instead, place an **AT** on the linkage instruction that the prior test was returning from when corruption was detected.
- If that linkage instruction is executed multiple times prior to the corruption happening, you may have to make the **AT** command conditional.
- Issue a **GO** command to let execution run until it reaches the linkage instruction at the right time.
- Now, issue **T** to let the linkage instruction transfer control into the suspect subroutine.
- Once you're in the subroutine, issue **T BNC (FALSE)** to restart the conditional trace, but this time within the suspect subroutine.

- Eventually, execution will stop when the Watch Conditionals again detect the corruption, and again execution will be either at a branching instruction or at a return from a deeper subroutine:

  - If execution is at a branching instruction, follow the process described above at step (5).

  - If execution is at a return from a subroutine, repeat the current process, step (6), but for this next subroutine.

- If you wind up having to repeat this step (6) frequently, then you may want to put your **TRACE WATCH** commands (and any other setup commands you may need) into a script and then run them with the **READ** command.

7) Sometimes a conditional trace can lose control. The reasons were mentioned above and are discussed in detail in **HELP BREAKPOINTS TRACING CONDITIONAL FAILURES.**

   Also, it may simply be the case that the starting point of your analysis is either too remote from the corruption or simply not the right place to start.

   It also might be the case that the Watch Conditionals are not correct for trapping the corruption that is actually occurring.

   Whatever the case, you will want to create some means by which you can terminate the conditional trace when it simply has gone on too long. There are a couple of ways to set that up.

   - **Defensive Traps:** If there are points in your code that will not be reached until after a corruption should have occurred, then you could use one or more **AT** commands to set traps at those points. Places to consider for such traps are:
     - Known exit points,
     - Known return points,
     - And places that you know execution will reach only after executing the logic of interest.

   Countdown Traps: You can set a countdown trap that will abort a conditional trace after Z/XDC as received control n number of times. For example:

   TRACE WATCH (10000)

   This sets up a Watch Conditional that simply counts the number of times Z/XDC has received control and the other Watch Conditionals have all resolved false. The trace will be aborted when Z/XDC is entered for the 10,000th time.

Help Breakpoints TRACing Conditional Performance

The Problem

Z/XDC's overhead while processing conditional traces is variable. It ranges from very high to utterly outrageous! This variation depends upon several factors, some of which are under the user's control.

Avoid Floating Maps and Equates

The existence of maps and equates degrades conditional trace performance either...
- Mildly for maps/equates with **assigned** bases,
- Or **SEVERELY(!)** for maps/equates having **FLOATING** bases!

The resolution of floating maps/equates is an intensive, repetitive and complex process. Every time Z/XDC receives control during a conditional trace, it has to **reresolve** the locations of **all** floating objects (even the ones that you might think don't matter!) in order to know how the conditional expression needs to be evaluated. Therefore, during conditional traces:
- Delete all floating objects except those that are absolutely necessary for resolving the conditional expression.
- If you have large numbers of fixed based maps and equates, consider reducing their numbers as well.

Close Unnecessary Watch Windows

Or make sure that no Watch Window contains any commands that cause address location resolution.

Note, **L PSW** is one such command. During conditional tracing, this (and similar) command can be extremely expensive!

Do Not Create an excessive number of Traps

Z/XDC maintains its breakpoint definitions in a queue that has to be searched linearly and completely each time Z/XDC receives control for conditional trace evaluation. This may affect performance when a very large number of breakpoints exist. When this is the case, consider removing breakpoints that are not immediately necessary.

Avoid Referencing Anything from a Map in Your Conditional Expressions

When a conditional trace is running, the conditional expressions involved are evaluated **every** time a recapture trap is reached. A complex conditional expression, of course, takes longer to evaluate than a simple one. Try to use expressions that are as simple as possible but still meet your needs.

Examples:

If you want to use a field name from a static map, consider using the address of that field instead. For example, suppose you want to check for corruption of SVC 201's slot in z/OS's SVC Table.

- If you have z/OS's SCVT mapped, you could use
  (.SCVTSVCT?+201N*8 NE whatever)

- But it would be more efficient to use
  (10?+c8?+84?+201N*8+201N*8 NE whatever)

- And even just a little bit more efficient to use
  (10?+c8?+84?+648 NE whatever)

- Even better, since the resolution of **10?+c8?+84?+201N*8+201N*8** isn't going to change, just find and plug in the table entry's actual address instead.

If you want to use a map that is floated off a register, then get rid of the floating map, and reference the register instead. Example:
(R9?+3BC EQ whatever)

Help Breakpoints TRACing Conditional Slipvsxdc

What is SLIP?

SLIP stands for "Serviceability Level Indication Processing". (Catchy, huh?). Ok, well its meaning may be... unfortunate, its capabilities are not.

SLIP is a powerful facility in z/OS that can be used to trap storage corruption problems. A **SLIP trap** can be activated:
- Via a System Operator command (the **SLIP** command), which you can issue:
  - From an Operator's Console,
  - From TSO's **OPER** command,
  - From IBM's **SDSF,**
  - From Triangle System's **IOF,**
  - Or from any other facility that offers Operator Console support.

The **SLIP** command is documented in the following IBM manuals:
- **z/OS MVS System Commands** (SA22-7627)
- **TSO/E System Programming Command Reference** (SA22-7793)

SLIP vs. Conditional Tracing

When trying to solve a storage corruption problem, you may be able to choose to use either z/OS's **SLIP** or Z/XDC's **conditional tracing.** Each has their advantages and disadvantages:

- The SLIP command is considerably more efficient than conditional tracing. The

performance degradation on your program is several orders of magnitude LESS for SLIP than for conditional tracing.

- Use of the SLIP command requires permission to use a System Operator's Console interface. Many data centers will not permit this to most users.

- The SLIP command can test for alteration of only pre-specified storage locations or ranges.

- With Z/XDC's conditional tracing, on the other hand, the location to be tested is reevaluated for each check. Thus, the test location can **dynamically react** to your running program's **realtime** actions.

- The SLIP command can test only for alterations in general. It cannot test for alterations to or from specific values or ranges.

- On the other hand, Z/XDC's conditional tracing can test for **selected** storage content changes.

- The kind of SLIP trap that checks for storage alterations is called a **PER trap.** z/OS permits only one PER trip to exist (system wide) at a time. (z/OS does have strategies to mitigate this limitation, that involve one PER trap triggering the next. So that's helpful.)

- Z/XDC allows you to create **multiple** WATCHs to test for as many conditions as you need, simultaneously.

- Storage Alteration SLIP traps are triggered the instant the monitored location is changed. Conditional expressions **(and this is important!)** cannot be evaluated and triggered unless and until Z/XDC receives control!

  Traps, traces and ABENDs cause Z/XDC to receive control, not conditional expressions. Conditional expressions, when they evaluate TRUE, will cause Z/XDC to **retain** control and present displays, but Z/XDC has to receive control in the first place in order for a conditional expression to be evaluated. (When all conditional expressions evaluate FALSE, Z/XDC will **silently** allow the interrupted program to resume.)

The Best Advantage of SLIP and the Best Advantage of Z/XDC

The best advantage of **SLIP** is its **performance.** Therefore, if the nature of your problem and politics are such that you can use SLIP, then I would strongly recommend that you do so.

The best advantage of **Z/XDC** is its **flexibility.** If the characteristics of your storage corruption problem are complex, then your best tool will be Z/XDC.

If you choose to use Z/XDC, and you run into performance problems, there are things you can do to mitigate those problems. See **HELP BREAKPOINTS TRACING CONDITIONAL PERFORMANCE** for suggestions.

Using SLIP Together with Z/XDC - Dumps

If you decide to use SLIP, you do not have to give up on using Z/XDC. It's not an either/or choice: You can still use Z/XDC in conjunction with SLIP.

Most times, SLIP is used to get a dump (ACTION=SVCD). Once you have that dump, you can use our **dump/XDC** Feature (together with IPCS) to help analyze that dump. See **HELP DUMPS** for more information.

Using SLIP Together with Z/XDC - Live Debugging

Alternatively, you can use SLIP and Z/XDC together to trap and debug a problem **interactively** on a live, running program! You do this as follows.

When you design a SLIP trap, one of the things that you can choose is what happens when the SLIP event occurs. (These are called "action options" in SLIP's documentation.) To make use of Z/XDC, the action choice that you should make is **RECOVERY.** This is IBM-speak for "ABEND the guy!".

The **ACTION=RECOVERY** SLIP command operand causes the user program to ABEND with code **s06F.** If Z/XDC has been established as the recovery routine for your program, then the **RECOVERY** action will eventually cause Z/XDC to receive control. Z/XDC's powerful displays can then be used to analyze, step through and solve the problem **live,** as the program is running.

Help Breakpoints TRACing Conditional Failures

Do Not Try T, T B and T BY Conditional Tracing!

T, T B and T BY traces will try to follow execution **everywhere!** Into subroutines, into IBM service routines, back out to calling routines. Everywhere! They do not stay local to the current routine. Consequently, these traces **are not suitable** for conditional tracing. Use **T BNC** instead.

Use Only T BNC, T SB and T SA for Conditional Tracing

T BNC, T SB and T SA traces do not follow linkage instructions into subroutines. They are designed stay local to the currently running routine or subroutine. Therefore, these traces **are suitable** for conditional tracing. See **HELP BREAKPOINTS TRACING CONDITIONAL** for more information.

Conditional Trace Failures - Missed Recapture Traps

This section is not about detected trace failures. When Z/XDC itself detects that it cannot continue with a trace, it will say so, and the resulting error message's Built-in Help **(HELP MESSAGES DBCnnnn)** will offer suggestions for what to do.

Unfortunately, it also is possible for traces to fail in ways that **Z/XDC cannot detect!**

An undetected conditional trace failure occurs when a predicted recapture trap **is missed** by program execution. When this happens, Z/XDC loses control of the trace and never gains it back unless some other trap is reached. (For details regarding Z/XDC's tracing methods as well as about recapture traps, see **HELP BREAKPOINTS TRACING CONDITIONAL.)**

Normally, Z/XDC's tracing is bullet proof. Briefly:

- Z/XDC has to accurately scan program code for branching instructions on which it sets temporary **recapture traps.**
- Then when user program execution reaches the trap, Z/XDC has to fully understand whatever branching instruction's been reached,
- So that it can accurately predict whether or not that instruction will, in fact, branch.
- If it is going to branch, then Z/XDC has to work out where the instruction will branch to.

So far, so good.

The potential for failures occurs at linkage instructions that a conditional trace has decided to **step over:**

- This happens for calls to subroutines located in **alien** storage.
- It also happens at **all** linkage instructions when a **T BNC, T SB** or **T SA** trace is being used.

This is because it is not possible to predict with absolute accuracy:

- Whether or not the subroutine will return,
- Whether or not the subroutine will, instead, take an exit to an unknown location.
- Or if the subroutine does return, what offset it will return to.

When stepping through code **manually,** these issues are not all that important, because the human can use his knowledge of the program's logic to be aware of non-standard situations and then to decide for himself where traps need to go so as to prevent loss of control. (If he doesn't yet have such knowledge, Z/XDC is a great tool for acquiring it!)

However, for **automagic** tracing (which is what conditional tracing is), it's a different story, because now Z/XDC has to predict for itself a future that cannot be predicted with absolute accuracy.

So Z/XDC makes assumptions that will be correct in **most** cases, but not all cases. These are discussed in detail in **HELP BREAKPOINTS TRACING SUBROUTINES,** but briefly:

- Z/XDC supports subroutine calls that may be followed by a branch vector that's up to six slots long. Z/XDC will successfully recapture execution if the subroutine makes a return to +0, +4, +8, +C, +10 or +14 past the linkage instruction. A return to any other offset will be missed.

- If the called subroutine is a standard z/OS **access method,** Z/XDC will find and set recapture traps at the **EODAD** and **SYNAD** exit points (if they exist). Z/XDC does not currently support any other exit points. (If you would like to make a suggestion, please contact us.)

Linkage Instruction Usage Analysis

When execution is located at a linkage instruction, Z/XDC may need to do an analysis to determine that instruction's usage. (See **HELP COMMANDS TRACE BNC** for details.) Typically, there are two distinct ways in which linkage instructions are used:

- **Call-with-return:** The instruction is being used to call a subroutine. If the subroutine is to be stepped over, then Z/XDC (if it can) needs to set recapture traps at the subroutine's return points.

- **Point-and-jump-over:** The instruction is being used to jump over data while loading a pointer to that data into the linkage registers. In this case, the trace needs to process the linkage instruction no differently from any other non-linkage branch: It needs to set the next recapture at or after the branch target.

This analysis is most accurate when the code containing the linkage instruction has been mapped.

When the code is not mapped, Z/XDC can still sometimes identify call-with-return with confidence, but if it can't, then Z/XDC takes the safe approach and follows the jump even when it might not otherwise need to. For full details, see **HELP COMMANDS TRACE BNC.**

Mitigations

If you want to kick off a conditional trace, do it using either **T BNC, T SB** or **T SA.** Do **not** use T, T I, T B and T BY.

The ok-to-use traces attempt to keep the trace within the local code. The not-ok-to-use traces do not. They will follow execution into as many subroutines as get called.

If you know that your trace is going to have to step over subroutines that make returns that Z/XDC does not support, or take exits that Z/XDC does not support, then before you start the trace, place defensive **AT** type traps:
- At known exit points,
- At known return points,
- And at places that you know execution will pass through after executing the logic of interest.

Conditional Trace Failures - Changes in the Recovery Environment

Breakpoints are either **X'00'** opcodes or **TRAP2** instructions. (See **HELP BREAKPOINTS TYPES** for details.)

- If breakpoints are **X'00'** opcodes, then Z/XDC must have previously been established as the current execution environment's **newest** recovery routine. Unpredictably **bad things happen** when that is not the case.

- If breakpoints are **TRAP2** instructions, then Z/XDC's Trap Handler must have been

setup for the current task. In order to set up the trap handler, Z/XDC needs to be the current execution thread's newest recovery routine at the time the **first** TRAP2 is reached by execution. After that, the recovery environment no longer matters.

You can use the **SET TRACE TRAP2|ZERO** to change the breakpoint's type at will. See **HELP COMMANDS SET TRACE TRAP2** for more information.

If a subroutine that's being stepped over creates ESTAEs that it doesn't also cancel, then when execution returns to the recapture trap:

- If the trap is an **X'00'** opcode, then the debugging session will fail, because Z/XDC no longer is the newest recovery, so your program's own recovery routine will see the s0C1 generated by the breakpoint's **X'00'** opcode, but he won't know what to do with it, so random bad things will happen.

- If the trap is a **TRAP2** instruction, then the change in recovery environment won't matter, so the execution recapture will be successful.

Monitoring a Long Running Conditional Trace

If you are running a conditional trace, and if you are getting bored and fidgety from staring at a locked keyboard, then if you want to see what's going on, then try the following:

- Use Z/XDC's Foreign Address Space Mode to monitor the progress of the conditional trace:
  - Switch over to another TSO user session.

  - Start an authorized Z/XDC debugging session (**XDCCALLA IEFBR14,** for example).

  - Use **SET ASID userid** to look at your locked TSO session's address space.

  - Use **LIST TASKS** to display that session's TCB tree.

  - Determine which TCB matters, then repeatedly use **LIST RBS tcb#n** to watch the progress of execution in your program.

- Or use the **ATTN** key to unlock the keyboard so that you can issue appropriate **LIST** commands. This can be done non-invasively, so when you are finished looking around, you can use the **GO** command to let your program resume as if nothing happened. See **HELP ATTENTIONS** for more information.

Help Breakpoints TYpes

Two Types of Breakpoints: s0C1 ABENDs vs. TRAP2 Instructions

A breakpoint can be implemented using one of two techniques, and the technique used can be different from one breakpoint to the next. The technique in effect is

displayed by the **LIST TRACE** command and set by the **SET TRACE ZERO|TRAP2** command:

- When SET TRACE **ZERO** is in effect, breakpoints are set by replacing the first byte of a machine instruction with **X'00'** (an invalid opcode). That replacement will cause a program check, and the resulting s0C1 ABEND may be intercepted by Z/XDC.

- When SET TRACE **TRAP2** is in effect, breakpoints are set by replacing the first 2 bytes of a machine instruction with **X'01FF'** (the TRAP2 instruction). If no TRAP handler has been installed for the current execution unit (Task or SRB), an s0D3 ABEND will occur and Z/XDC may see it. If it does, then Z/XDC will install a TRAP handler and then resume your program to re-execute the TRAP2 instruction.

  After the Z/XDC TRAP handler is installed, TRAP2 instructions will cause Z/XDC to receive control as a Trap Handler for processing the breakpoint. Note, this eliminates the System's ABEND processing overhead of fielding a System Interrupt, of running the Recovery/Termination Manager and of scheduling Z/XDC as an ESTAE routine.

Things to Notice About TRAP2-Type Breakpoints

A Trap Handler is always first in line, ahead of all other forms of Interrupt processing. This means that once a Trap handler Exit is established, TRAP2-type breakpoints will never be seen by:

- ESTAE-type ABEND recovery routines (ESTAEs, ESTAEXs and ARRs),
- FRRs
- **SPIEs and ESPIEs** (program check exits)

The distinction between an **error level** environment and a **retry level** environment simply disappears! The two environments are always one and the same.

There is a small probability of a Trap Save Area overlay or corruption when using TRAP2-Type breakpoints.

There are some restrictions with Z/XDC's support of TRAP2-type breakpoints pertaining to SRBs. These are:

(1) SRBs will never have a TRAP handler installed, so TRAP2 instructions will always cause an s0D3 ABEND. But TRAP2-type breakpoints can still be used because Z/XDC has been changed to process an s0D3 as if it were a regular breakpoint.

(2) An FRREND exit will not be driven if an END command is issued when a TRAP2-type breakpoint is being processed on behalf of a task in a restricted mode (when an FRR would have been driven).

Z/XDC creates a breakpoint in a user's program by changing a target machine instruction's first one or two bytes...

- Either to **X'00'** (an illegal opcode),

- Or to **X'01FF'** (a TRAP2 instruction).

So when user program execution eventually reaches that instruction...

- Either a **0001 program check interrupt** occurs which will eventually cause Z/XDC to receive control from the System's Recovery/Termination Manager (RTM) with an indication that an s0C1 ABEND has occurred,

- Or a **Trap Exception** occurs which will cause the hardware immediately to pass control to a Z/XDC-owned Trap Handler Routine which will very quickly pass control to Z/XDC proper.

The trapping method used is controlled by the **TRAP2** and **ZERO** operands of the **SET TRACE** command. See HELP COMMANDS SET TRACE TRAP2 for more information.

Breakpoints via s0C1 ABENDs

In order for the s0C1 method to work, Z/XDC **must** be set up as your program's **newest ABEND Recovery** routine (ARR, ESTAE, ESTAEX, ESTAI, etc.). Also, a SPIE or ESPIE **must not exist!**

Then when the 0001 program check occurs, the program is not terminated. Instead, the resulting s0C1 ABEND is intercepted, and the System passes control to Z/XDC as an ESTAE-type (or FRR-type) ABEND Error Recovery routine.

Breakpoints via TRAP2 Instructions

On the other hand, if the TRAP2 method is being used, then a Z/XDC-owned Trap Handler Routine receives control directly from the hardware, and so RTM and other System overhead is avoided.

In this case, Z/XDC is running as a Trap Handler Routine (not as an ABEND Recovery Routine):
- ESTAE, ESTAEX and ARR routines do not receive control.
- FRRs do not receive control.

- **SPIEs and ESPIEs do not receive control!**

Why Z/XDC Still Must be an ABEND Recovery Routine

At first blush, when TRAP2-type breakpoints are being used, it might seem that making Z/XDC your program's newest ABEND Recovery Routine would not be necessary. **But that is not actually the case.** This is because in order for TRAP2 instructions to work, a Trap Handler Routine must be installed for each execution thread (Task or SRB) to be debugged.

When such a Handler is not installed, the TRAP2 instruction does not work. Instead, a Special Operation Exception occurs (program interrupt code 0013), and that eventually leads to an s0D3 ABEND.

But if Z/XDC is set up as your newest ABEND Recovery Routine, it will see that s0D3 ABEND, and it will be able to recover from it! It does this by **dynamically**

installing the missing Trap Handler and then resuming execution to reattempt the TRAP2.

Once the Trap Handler Routine is in place, Z/XDC no longer needs to be your program's newest Recover routine... at least not for the purposes of trapping and tracing your code. **But...**

- It will still be needed to capture any other ABENDs that your program may suffer.
- And for multi-threaded programs, it will still be needed for setting up Trap Handler Routines in any other of your tasks (and SRBs) that you may wish to debug.

Error Level and Retry Level Execution Environments Considerations

When an ABEND occurs, The System's Recovery/Termination Manager (RTM) gains control and passes control to an ABEND Recovery Routine (such as Z/XDC), and he provides to the Recovery Routine (Z/XDC) information about two distinct environments:

- The **Error Level** environment consists of the registers, PSW and state data (AMODE, ASC-MODE, execution state, Request Block queue depth, etc.) **that existed** at the moment that the program ABENDed.
- The **Retry Level** environment consists of the registers, PSW, state data and Request Block level that **will exist** in the event that the Recovery Routine tells RTM to allow the ABENDed program to resume execution.

There are many very good reasons why these two environments are not the same. For details, please read HELP EXECUTIONLEVELS.

But remember! The distinction between the Error and Retry Levels is an **artifact** of the ABEND and recovery process. So it arises for 0C1-type breakpoints. It does **not** arise for **TRAP2-type** breakpoints!

Error vs. Retry Level Considerations Arising from 0C1-Type Breakpoints

When Z/XDC is in control due to an **0C1-type** breakpoint, the Error Level and retry Level environments may be the same, or they may be different. When they are **different,** the usefulness of Z/XDC is limited:

- When you use the **LIST RBS** command, The Error Level Request Block will be separately captioned.
- You have to use the **EWHERE EPSW?-2** command to view the Error Level execution location.
- You can use the **LIST ERWREGS** command (for example) to view the Error Level general registers.
- You will **NOT** be able to use the **TRACE** command to step execution through Error Level code. Only Retry Level code can be stepped through.
- You will **NOT** be able to use the **GO** command to resume Error Level execution. Retry Level execution will be resumed (not Error Level execution).

On the other hand, when the Error Level and Retry Level environments are **the same,** everything is much better:

- When you use the **LIST RBS** command, The Error Level Request Block will not be separately captioned. Only the Retry Level Request Block will be labeled.
- You can use the **WHERE** command to view the current execution location.
- You can use the **LIST RWREGS** command (for example) to view the Retry Level general registers.
- You **will** be able to use the **TRACE** command to step execution through your code.
- You **will** be able to use the **GO** command to resume your program's execution.

So usually, you would want to strive to make sure that the Retry Level and Error Level execution environments are the same in any code that you wished to debug. For information on how to do that, see HELP EXECUTIONLEVELS.

Error vs. Retry Level Considerations Go Away for TRAP2-Type Breakpoints (Almost)

When Z/XDC is in control due to a **TRAP2-type** breakpoint, there is no distinction between the Error Level and Retry Level environments...

- All of Z/XDC's Error Level constructs (EWHERE command, ERn register names, etc.) will still work, but they will display generally the same information as the Retry Level constructs.

  However, the two sets of displays will diverge when:
  - You use the **ZAP** command to change register contents,
  - Or you use the **SET PSW** command to change things in the PSW.

  All such changes will be reflected in the various **retry level** state constructs and displays. The error level state data will remain unchanged. They will continue to show things as they were at Trap time.

- A **LIST RBS** command will not show any **(ARTIFACT)** RBs:
  - There will be no **PGM CHK** RB for running the RTM (IGC0101C),
  - There will be no **SYNCH** RB for running Z/XDC.
  - The only Request Blocks that will be shown are those that actually belong to the program being debugged (and whatever System Services it may have invoked).
- All Retry Level data (registers, PSWs and states) will initially be **identical** to the Error Level data (except the Retry Level PSW will have been adjusted backwards by two bytes).
- You will be able to use the **TRACE** command freely to step through your program's execution.
- You will be able to use the **GO** command freely to resume your program's execution.

A Nomenclature Issue

For historical reasons, we have gotten into sort of a bind over exactly what is meant by the word **trap:**

- On the one hand, it has a functional meaning concerning the differences between breakpoints that are used to assist in the stepping of execution through your code, vs. those that are intended to trap program execution at a specific location.
  - I will refer to the former as **tracing breakpoints** because they are created, originally by the **TRACE** command (but more recently also by c/XDC's **STEP** command).
  - I will refer to the latter as **trapping breakpoints** because they are used to trap execution at specific locations. They are created by the **TRAP AT ATX TDEFERRED** and **HDEFERRED** commands.
- While on the other hand, the word **trap** has an implementation meaning regarding the physical nature of the breakpoint itself. Specifically, breakpoints may be built from:
  - Illegal opcodes **(X'00')** which I'll refer to herein as **0C1-type** breakpoints,
  - Or TRAP2 instructions **(X'01FF')** which I'll refer to herein as **TRAP2-type** breakpoints.

Both tracing and trapping breakpoints can be implemented either as 0C1-type breakpoints or TRAP2-type breakpoints. So to recap:

- **TRAP2-type breakpoint** refers to any breakpoint (for tracing or for trapping) that is implemented via a TRAP2 instruction.
- **Trapping breakpoint** refers to any breakpoint (regardless of implementation) that is intended to trap program execution at a specific location. Trapping breakpoints can be built from either 0C1-type breakpoints or TRAP2-type breakpoints.
- **Tracing breakpoint** refers to any breakpoint (regardless of implementation) that is used by the **TRACE** command (and **STEP** command) for stepping execution through your code.
- **0C1-type breakpoint** refers to any breakpoint (for tracing or for trapping) that is implemented via an **X'00'** "opcode".

Help Breakpoints TYpes Corruption

TRAP Save Area Corruption

Under a task in z/OS, there is a small possibility of a Trap Save Area overlay or

corruption occurring.

This can happen because there is one Trap Save Area per dispatchable unit. In most cases a dispatchable unit in hardware terms corresponds to a dispatchable unit in software terms. However, this is not the case with z/OS. When a task is executing, that execution is represented by a Request Block (RB). Execution of the RB can be interrupted at any time and a new RB, called an Interruption Request Block (IRB) can be executed. The IRB and the (now suspended) RB are using the same Trap Save Area and it is possible that the information saved by the hardware (when the TRAP2 instruction executes) could be overlaid by newer information (if there is a TRAP2-type breakpoint in the IRB) before the trap handler running under the RB has saved the information.

When this occurs Z/XDC detects the corruption or overlay and will display a message warning you of the corruption or overlay.

Also, Z/XDC will not allow you to resume execution of the program unless you use the FORCE operand of the GO and TRACE commands. You will first need to alter the values in the (retry) general registers, AR15, and PSW resume address (using the **ZAP** command) and the PSW AMODE, ASC-MODE, and Condition Code (CC) (using the **SET PSW** command).

For more information on the **GO** command, see **HELP COMMANDS GO**
For more information on the **TRACE** command, see **HELP COMMANDS TRACE**
For more information on the **ZAP** command, see **HELP COMMANDS ZAP**
For more information on the **SET PSW** command, see **HELP COMMANDS SET PSW**

Help CHaractersets

Z/XDC does most of its internal processing using EBCDIC. Specifically, Z/XDC uses Code Page 1047 (CP1047) internally.

When processing commands From all sources:
- a command line
- READ file input lines
- command strings queued by REXX

a translation of the code points for the square brackets is done so that these sources can also use CP037 and CP500 code points for them.

In CP1047, a left square bracket’s code point is X’AD’, and a right square bracket’s code point is X’BD’.

In CP037 and CP500, a left square bracket’s code point is X’BA’, and a right square bracket’s code point is X’BB’.

Occasionally, Z/XDC interprets byte values in ASCII. When it does this, Z/XDC uses ISO-8859-1.

Some Z/XDC commands have the EBCDIC or ASCII operands. These operands can be used to control how Z/XDC interprets character data. These commands include:

- **DISPLAY**
- **EWHERE**
- **FORMAT**
- **LIST GENERALREGISTERS REGISTERSET**

- **SET FORMAT**
- **WHERE**

When using c/XDC, sometimes a variable's value is shown as an EBCDIC or ASCII string. These commands include:

- **LIST VARIABLES**

When displaying in EBCDIC, bytes with the following hexadecimal values are displayed as the character:

high	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x		.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
1x		.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
2x		.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
3x		.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
4x			.	.	.	.	.	.	.	.	?	.	<	(	+	|
5x	&	.	.	.	.	.	.	.	.	.	!	$	*	)	;	?
6x	-	/	.	.	.	.	.	.	.	.	?	,	%	_	>	?
7x		.	.	.	.	.	.	.	.	?	:	#	@	'	=	"
8x		.	a	b	c	d	e	f	g	h	i	.	.	.	.	.
9x		.	j	k	l	m	n	o	p	q	r	.	.	.	.	.
Ax		.	~	s	t	u	v	w	x	y	z	.	.	.	[	.
Bx		.	.	.	.	.	.	.	.	.	.	.	.	.	]	.
Cx	{	A	B	C	D	E	F	G	H	I	-	.	.	.	.	.
Dx	}	J	K	L	M	N	O	P	Q	R	.	.	.	.	.	.
Ex	\	.	S	T	U	V	W	X	Y	Z	.	.	.	.	.	.
Fx	0	1	2	3	4	5	6	7	8	9	.	.	.	.	.	.

(Some characters cannot be displayed in the above matrix, due to limitations in the Z/XDC help system. These characters are shown as '?' in the above matrix and their value is:

X'4A' - cent sign
X'5F' - not sign
X'6A' - broken vertical bar
X'6F' - question mark (?)
X'79' - oblique)

(Note that, although Z/XDC uses CP1047 internally, when displaying an EBCDIC translation, some byte codes that translate to a valid character are displayed as '.' to ensure that the resulting display is valid for all terminals)

When displaying in ASCII, bytes with the following hexadecimal values are displayed as the character:

high	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x		.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
1x		.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
2x			!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	.	_
6x	?	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	.
8x		.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
9x		.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
Ax		.	.	?	.	.	.	?	.	.	.	.	.	?	.	.
Bx		.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

```
 Cx    .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 Dx    .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 Ex    .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
 Fx    .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
```

(Some characters cannot be displayed in the above matrix, due to limitations in the Z/XDC help system. These characters are shown as '?' in the above matrix and their value is:
  X'3F'   - question mark (?)
  X'60'   - oblique
  X'A2'   - cent sign
  X'A6'   - broken vertical bar
  X'AC'   - not sign)

(Note that, although Z/XDC uses ISO-8859-1 internally, when displaying an ASCII translation, some byte codes that translate to a valid character are displayed as '.' to ensure that the resulting display is valid for all terminals)

Help Hooks

A **hook** refers to a process wherein a small piece of a program's code is changed (during execution) such that when that code is executed, control is suddenly transferred to another program. Generally, when that other program finishes whatever it was written to do, control returns to the interrupted program which then should be able to proceed as if nothing has happened. Essentially, the hook occurs without the knowledge or cooperation of and without interference with the interrupted program.

A Z/XDC Hook is a hook that causes a Z/XDC debugging environment to be created and that causes execution to be passed to Z/XDC so that a debugging session with a user can be created or continued.

As discussed at length in HELP DEBUGGING GETTINGSTARTED, Z/XDC is a recovery routine (ESTAE, ESTAEX, FESTAE, ARR or FRR). This is the root of both its great power and its greatest weakness. Because Z/XDC is a recovery routine, it can be used in a very large number of z/OS environments. However, also because it is a recovery routine, its usefulness can be defeated by the existence of other recovery routines that may have been created by the program (or system service routines, or exit routines) that you may want to debug.

Basically, in order for Z/XDC to be effective, it must always be the **newest** recovery routine for the code that is being debugged. If for example, user code has issued it's own ESTAEX for the local code, then a Hook will need to be set following that ESTAEX in order to make Z/XDC the newest recovery routine again.

The several sections of the Built-in Help named "HELP DEBUGGING ESTAES ..." discuss several approaches and considerations for dealing with an existing error recovery environment and with integrating Z/XDC in various ways into such environments. Another approach, discussed here, is to create a Z/XDC hook.

Z/XDC provides several methods for inserting a Z/XDC hook into a program.

The HOOK command

The HDEFERRED command
These are commands that can be used to set **dynamic** hooks into many kinds of programs running under any task in any address space (as permitted by System Security). A hook can be inserted at nearly any point of a program's code. When the hook is reached by program execution, special code is executed that:
- Removes the hook by restoring the program's original code to the hooked location.
- Adjusts the program's PSW so that the restored code will be executed.
- Builds a STAE Control Block (SCB) or FRR that makes Z/XDC the newest recovery routine owned by the program.
- Detects whether or not an (E)SPIE exists that intercepts program checks 0001 thru 0007 (ABENDs 0C1s thru 0C7s). If such is found then a **ESPIE RESET,=A(0)** is issued to delete the (E)SPIE entirely.
- Passes control to Z/XDC so that it can display the intercepted program and process your commands.

The **HOOK** command can be used to place a hook into code that already resides in storage.

The **HDEFERRED** command is used to place a hook into a load module that has not yet been loaded into storage. The hook can be targeted to any location within a load module (not just the module's entry address). (See HELP HOOKS DEFERRED for more information.)

The #XDCHOOK macro
This is a macro that can be used to assemble a **static** hook into a program. The macro generates code that scans through certain system and Z/XDC control blocks to find Z/XDC's hook support routines, which it then BASRs to. When reached by execution, this code performs all of the same functions described above for dynamic hooks **except** it does not remove the hook code from the user program. It just leaves the #XDCHOOK macro generated code intact.

The difference between a hook and a breakpoint is this:
- A **breakpoint** requires a Z/XDC debugging session to be already established before the breakpoint is reached by execution. (If it is not, then the breakpoint is just an ordinary s0C1 as far as the System is concerned.)
- A **hook,** on the other hand, **creates** a debugging session when one did not previously exist.

Subtopics Index

The following topics explain various aspects of creating and using Z/XDC hooks. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

DYNAMIC - Hooks created dynamically using the HOOK command.
STATIC - Hooks created statically at assembly time.
DEFERRED - Hooks targeting load modules that have not yet been loaded into storage.
LEGACYSUPPORT - Continuing support for the HOOK SVC arising from use of legacy versions of the #XDCHOOK macro.
UNDOING - How to remove an ESTAEX or FRR (for Z/XDC) created by a dynamic

hook.

For related information, please see the following topics:
- HELP COMMANDS HOOK
- HELP COMMANDS HDEFERRED
- HELP COMMANDS LIST HOOKS
- HELP COMMANDS DELETE HOOKS
- HELP EXECUTIONLEVELS

Help Hooks DYnamic

A dynamically created Z/XDC hook is one that is created in a currently running program "on the fly" by Z/XDC's HOOK command. Such hooks are designed to create (or resume) debugging sessions without altering any user program register or state attribute (AMODE, ASC-MODE, cross-memory status etc.).

The HOOK command can be used to target most code running in any address space (security permitting), including the Home Address Space. A dynamic hook is a convenient way to insert a Z/XDC debugging session into code both regardless of the error recovery environment within which that code runs, and without having to make any source code changes to that program.

Hooks can be inserted into nearly any kind of code running pretty much anywhere. This includes:
- Problem mode code
- Authorized code
- Taskmode code
- SRB routines
- ESTAE protected, ESTAI-protected, ARR-protected, etc-protected code
- FRR protected code
- PC routines
- Locked code
- Disabled code
- PRIMARY and AR ASC-MODE code
- SECONDARY AND HOME ASC-MODE code
  (See below)
- Code running in Foreign Address Spaces
- Whatever!

(If you want to set a hook into code could be executing in secondary or home ASC-MODE, you must nominate a work register and indicate that the code may be in these ASC-MODEs).

Setting the Hook (HOOK Command Processing Summary)

When you use the HOOK command, you must give an address expression that identifies the location and address space at which you want the hook placed. Z/XDC then does the following:

- It obtains storage in an appropriate subpool (which varies with the expected execution environment), and it builds a control block that describes the hook that is about to be set. This control block is called a **HOOKCBE.**

- It saves **several bytes** of the targeted program's code (starting at the **hook point)** into the HOOKCBE. This is so that the program code can be restored and executed when the hook code is reached (by execution) and removed. (The exact number of bytes saved and replaced depends on several things).

- It also builds the prolog portion of Hook's support code into the HOOKCBE.

- It then zaps the hook point, storing there an instruction sequence that jumps to the prolog code in the HOOKCBE.

- The HOOK command assigns a name to the hook. The name is of the form **HOOKnnnn.** Note, this name is **not** an an equate and, therefore, cannot be displayed by the LIST EQUATES command. It can be displayed only by the the **LIST HOOKS** command.

- The HOOK command creates information within Z/XDC's data structures (in the local address space) so that Z/XDC can remember that the hook exists and so that it can display the hook via the **LIST HOOKS** command.

Displaying Hooks (or Maybe Not)

It is important to keep in mind that hooks, once created by the HOOK command, exist and survive independently of the debugging sessions that created them. It is possible to start a debugging session, place a hook into program code located in the current address space's private area, in another address space's private area, or in common storage, and then terminate that debugging session. The hook will survive for as long as the program containing the hook remains in storage and remains unexecuted. (Dynamic hooks are automatically removed when they are executed.)

On the other hand, Z/XDC's knowledge of the **existence** of a dynamic hook dies when the creating debugging session ends. This has the following consequences:

- The LIST HOOKS command can display only those dynamic hooks that were created during the current debugging session.
- The LIST HOOKS command cannot display dynamic hooks that were created either by other debugging sessions or by prior debugging sessions.
- The name of a hook does not survive past the debugging session in which the hook was created. As far as other debugging sessions are concerned, the hook is unnamed.
- The LIST HOOKS command cannot display static (i.e. assembled) hooks at all.

- The FORMAT command might not accurately format hooks that were not created in the current debugging session.

- The DELETE HOOKS command **CAN** delete dynamic hooks created in any debugging session, but if the hook was not created in the current debugging session, then **YOU** have to already know where the hook is.

Subtopics Index

The following topics discuss the placing of dynamic hooks into various areas of storage. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

SECURITY - Security and accessibility issues regarding the placing of dynamic hooks using the HOOK command.
HOMESPACE - Placing hooks into the private areas of the Home Address Space.
OTHERSPACES - Placing hooks into the private areas of other address spaces.
COMMONSTORAGE - Placing hooks into programs located in common storage.
WHATHAPPENS - What happens when execution reaches a Dynamic Hook.

The **GO REMOVEXDC** command can be used to remove ESTAEX and FRR routines created by Static and Dynamic hooks. For more information about this, see HELP HOOKS UNDOING.

Help Hooks DYnamic Security

Z/XDC's HOOK command can be used to set a dynamic hook into nearly any code located in any program located in any address space, including the Home Address Space. Whether or not you actually can (or should) place a hook at a particular location depends upon the following:

- The location's store-accessibility to you.
- Whether or not you have been given "zap" authority to that location by System Security.

Specifically, when you want to use the HOOK command to set a hook, the following access and security considerations apply:

ACCESS CONSIDERATIONS:

Z/XDC does not necessarily have to be running authorized before the HOOK command can be used. Whether or not it does have to be authorized depends on the accessibility of the targeted hook point. When Z/XDC is authorized, it is capable of placing a hook at any private or common storage location in any address space, including so called "Store Protected" areas of storage such as the PLPA, the read-only sections of the System Nucleus, among others. (Whether or not Z/XDC will actually let you place a hook an any or all of these areas depends upon System Security rules discussed below.) On the other hand, when Z/XDC is running non-authorized, only those areas of storage that have been assigned protection keys 8 or 9 can be hooked (again, depending upon security rules). Thus, when Z/XDC is **non-authorized** the following accessibility limitations apply:

- Hooks can be placed only in the Home Address Space, not in foreign address spaces.

- Although it is possible for some areas of common storage to be assigned protection keys 8 or 9, it is rare that that happens. So for all realistic purposes, hooks cannot be placed in common storage.

- Programs that are **both** reentrant (RENT) and that reside on disk in authorized libraries are loaded by the System into key 0 storage (not key 8 storage like other programs). Therefore, hooks cannot be placed in such programs. Note, it is not relevant whether or not such programs themselves are authorized. What is relevant is whether or not they reside in authorized

libraries. For more information about this problem and a real good suggested solution, see HELP DEBUGGING REENTRANT.

- On systems that have the **REFRPROT** facility both installed and activated, programs that are refreshable (REFR) are loaded by the system into key 0 storage (regardless of the load library's authorization level and regardless of the RENT attribute). Therefore, hooks cannot be placed in such programs without first turning REFRPROT off. For more information about dealing with this problem, see HELP DEBUGGING REFRESHABLE.

- When programs are loaded into key 9 storage or into user key storage (usually key 8 storage), Z/XDC is capable of placing hooks into such programs.

SECURITY CONSIDERATIONS:

Regardless of whether or not Z/XDC is running authorized, Z/XDC first checks System Security to see whether or not the current user has zap authority to the desired hook point. The rules that Z/XDC checks are the following:

- XDC.AUTH
  If Z/XDC **is** running authorized, then it first checks to see if the user has "read" access to the security rule named "XDC.AUTH". See HELP SECURITY AUTH for more information.

- XDC.FASM.jobname
  If the hook is being placed in another address space, then Z/XDC next checks with System Security to see whether or not the user has "alter" authority to the appropriate instance of this rule. See HELP SECURITY FASM for more information.

- XDC.ZAP.location-description
  If the above security checks are passed (or if they were not made), then Z/XDC next checks with System Security to see if the user has "alter" authority for the particular "XDC.ZAP.---" rule that best describes the storage to be hooked. (In the HOOK command's case, this would usually, but not necessarily, be a rule whose name is "XDC.ZAP.modulename".) See HELP SECURITY ZAP for more information.

Help Hooks DYnamic Homespace

A dynamic hook is a convenient way to insert a Z/XDC debugging session into nearly any code both regardless of the error recovery environment (ESTAE-type or FRR) within which that code runs, and without having to make any source code changes to that program.

Z/XDC's HOOK command can be used to insert a dynamic Z/XDC hook into "any" address space (security permitting), including the Home Address Space (the address space within which Z/XDC itself is running). Note, when debugging a batch program via cdf/XDC, the Home Address Space is the batch job's address space, not the TSO session that you might be using to control the debugging session.

The HOOK command can be used to insert a hook into a program running in the Home Address Space. That's all well and good, but now the question arises, "Why would you want to do that?". After all, if you're able to use the HOOK command in the first place, then a debugging session must already be running. Well, here are some situations in which the HOOK command can be very helpful.

- When Z/XDC is used to debug a program, a problem arises when that program happens to issue ESTAE(X), SETFRR or similar macros to create its own ABEND recovery routine. Such recovery routines interfere with Z/XDC processing. See HELP DEBUGGING ESTAES for more information about interfering error recovery environments.

- A similar problem arises if you want to debug certain kinds of exit routines: attention exits, timer exits, DCB exits, etc. These exits are "Request Block" driven. This means that they execute under the control of Request Blocks that are different from the Request Block under which your program's main code executes. This fact means that unless an ESTAE macro for Z/XDC is coded within the exit routine itself, Z/XDC cannot be used to trace the flow of execution through such routines. See HELP EXECUTIONLEVELS for more information about this.

- You may want to initiate a debugging session in a subtask that is running outside of the scope of the current debugging session. Example:

  - Suppose that you have started a debugging session in a program that is running within ISPF. Then the scope of the debugging session is limited to that program and its descendant tasks. ISPF tasks generally are ancestors of the program and so are outside of the scope of the debugging session. You could use the HOOK command to hook a debugging into the an ISPF task (such as ISPMAIN or ISPTASK), but be careful! Usually, ISPF load modules are located in common storage, so an attempt to hook such a module could trigger a debugging session in another TSO user's address space instead of your own! To avoid this, you would have to take steps to load copies of ISPF load modules into your own private area and then target them with the hook command instead of the common storage copies. See HELP DEBUGGING ISPF DIALOGS for more information.

When using the HOOK command to hook a program that is running in the Home Address Space, the following considerations apply:

- Z/XDC does not necessarily have to be authorized in order to use the HOOK command. If the hook target is writable for non-authorized programs, then authorization is not necessary. See "Access Considerations" in the "HELP HOOKS DYNAMIC SECURITY" topic for more information.

- Regardless of whether or not Z/XDC is authorized, Z/XDC always checks System Security to see if an XDC.ZAP.--- rule exists for the targeted location, and to see if you (the user) have "alter" access to that rule. See "Security Considerations" in the "HELP HOOKS DYNAMIC SECURITY" section for more information.

- When program execution reaches the hook point, the hook will be immediately cleared. Therefore, a hook can create, at most, only one debugging session.

- When program execution reaches the hook point, either the current debugging session will be continued or a new debugging session will be created depending upon whether the task that executed the hook is or is not a descendant of the

task that controls the original debugging session.

Help Hooks DYnamic Otherspaces

A dynamic hook is a convenient way to insert a Z/XDC debugging session into nearly any code regardless of the error recovery environment within which that code runs and without having to make any source code changes to that program. Also, the program being hooked does not even need to be running in the same address space as Z/XDC. It could be in nearly any address space in the System (security permitting. See HELP HOOKS DYNAMIC SECURITY for details.)

When using the HOOK command against a foreign address space, the following considerations apply:

- Your computer's Security System must not have rules that prevent you from zapping the target location in the Target Address Space.

- If program execution reaches a hook in a foreign address space, Z/XDC will receive control in that address space and will attempt to initiate an interactive conversation with a user as follows:

  - If the address is anything but a TSO address space, then Z/XDC will attempt to connect to cdf/XDC and wait for a programmer to signon to the debugging session. In order for this to work, the **XDCSRVER** Server Space must be up and running. See HELP XDCSRVER CDF for more information.

  - Z/XDC's attempt to use cdf/XDC can be overridden by the presence of a **//XDCWTOR DD DUMMY** JCL card. When this is present, then the debugging session will use WTO/WTORs for interactive communication with the programmer. See HELP USERINTERFACES WTOWTOR for more information.

  - Otherwise, if the targeted address space is a TSO session, then Z/XDC will establish communications with the user's terminal. This can be overridden by the presence of either of the following dummy allocations:

    ALLOC FILE(XDCWTOR) DUMMY REUSE
    Z/XDC will use WTO/WTORs to communicate with the programmer via System Operator Consoles.

    ALLOC FILE(XDCCDF) DUMMY REUSE
    Z/XDC will attempt to use cdf/XDC for communicating with the user. This is sometimes necessary when debugging ISPF dialogs to avoid fights between Z/XDC and ISPF over who does and does not own the TSO terminal. (See HELP DEBUGGING ISPF for more information.)

    ALLOC FILE(XDCWTOR) DUMMY REUSE
    ALLOC FILE(XDCCDF) DUMMY REUSE
    When **BOTH** allocations are present, Z/XDC will first attempt to communicate via cdf/XDC. If that fails, then it will try to use WTO/WTORs as a fallback. If that too fails, then Z/XDC will give up and percolate the ABEND.

One particularly useful thing that can be done with Dynamic Hooks is to start a brand new debugging session in an already running batch job or started task. See HELP HOOKS DYNAMIC OTHERSPACES STARTINGNEWSESSIONS for more information.

Help Hooks DYnamic Otherspaces Startingnewsessions

One particularly useful thing that can be done with Dynamic Hooks is to start a brand new debugging session in an already running batch job or started task. The nice thing is that this requires no preparation on the part of the job or task. All you have to do is generally this:

- Start an authorized debugging session using **XDCCALLA IEFBR14.** This has nothing to do with the target aspace in which you want to start your actual debugging session. This is just a **token** session from which you can issue the authorized commands needed to start the session you actually want.

- Use the **SET ASID** command to switch into Foreign Address Space Mode (FASM) targeting the address space in which you want to start your debugging session.

- Use various commands to locate the code point at which you want a debugging session to start:
  - You can use **LIST TASKS** to see what tasks are running.
  - You can use **LIST RBS TCB#n** (or the **L** shortcut command) to see what programs and services are running under a task of interest.
  - You can use **MAP address** (or the **M** shortcut) to help identify where csects are within a module and what statement labels are within a csect.
  - You can use **FORMAT address** (or the **F** shortcut) to display the code of interest.

  Notice the variety of **shortcuts** that can be used in the shortcut input fields. They often are an easier way to enter many of these commands.

- Once you’ve figured out where you want to set a hook, use the **HOOK address** command (or **K** shortcut) to set the hook.

- Now do whatever is necessary to cause your program’s execution to reach the hook point. When it does, the following things happen:
  - Hook support first removes the hook from the code by restoring the original

logic to the hook point.

- Hook support then issues an **ESTAEX** or **SETFRR** to make Z/XDC the newest recovery routine in the hook point's environment.

- Hook support then sets a trap at the hook point and returns execution to the hook point, thus immediately executing the trap.

- The trap finally causes Z/XDC proper to get control. So he looks around and very quickly discovers that he needs somebody to talk to.

- So Z/XDC reaches out to **cdf/XDC,** and he goes into a wait state waiting for you to connect into the session.

- So now **you** need to:
  - Logon to a **TSO** terminal,
  - Start up **ISPF,**
  - Navigate to the **Z/XDC Startup Panel,**
  - Type a **3** in the command line and press ENTER.

  This starts a small piece of communications software called **slu/XDC.** It knows how to find, display and connect to pending debugging sessions.

  You could use the same TSO session in which you ran the token session, but you'd have to end that session before you'd be able to navigate over to the Z/XDC Startup Panel. It may be more convenient to logon to an entirely new and separate TSO session.

- slu/XDC finds all pending debugging sessions that you are allowed to connect to, and it builds/displays a **Job Selection Menu.** You then type an **S** at the left, and voila! You're connected to the session and you're off and running.

Help Hooks DYnamic Commonstorage

A dynamic hook is a convenient way to insert a Z/XDC debugging session into nearly any code regardless of the error recovery environment within which that code runs and without having to make any source code changes to that program. Dynamic hooks can be inserted into programs that reside either in any address space's private area or in common storage. When a hook is set into a common storage program, special concerns arise and need to be discussed.

When Z/XDC is running authorized, its HOOK command can, in principle, be used to set a hook into any code located anywhere within common storage **regardless** of whether or not the code resides in "page protected" storage (the PLPA, read-only areas of the System Nucleus, and sometimes the MLPA and the FLPA). However, before setting such a hook, Z/XDC first checks with System Security to see whether or not the user has "zap authority" to the location that he is targeting. See HELP HOOKS DYNAMIC SECURITY for more information.

Obviously, you must be extraordinarily careful when setting hooks into common storage routines. This is because common storage routines can be executed by any address space in the System, and because common storage routines often are system critical. A misplaced hook can have such undesirable results as:

- Creating Z/XDC debugging sessions in another user's address space.
- Crashing major subsystems such as CICS, VTAM, JES2, IMS, DB/2 (to name just a few).
- Crashing the entire system in less time than it takes you to remove your finger from the ENTER key!
- Crashing the System hours, days, or even weeks later when everyone has moved on to other projects and no longer has any recollection of what they might have done.

For more information relating to debugging PLPA and Nucleus programs, see:

- HELP SECURITY
- HELP DEBUGGING PLPA

Common storage is, of course, shared to all address spaces in the System. This means that when the contents of a location in common storage are changed, that change is instantly visible to all address spaces in the System. In particular, if a hook is set into a common storage program, then that hook could be executed (and in many cases **will** be executed) by any and every address space in the System! Therefore, it is very important for you to be very careful when using the HOOK command against a common storage program.

When the HOOK command sets a hook into a either a common storage location or a private storage location that is with a space switching PC routine, you need to associate with that hook a targeted address space. This is so that when the hook is executed, the Hook Service can determine whether or not it is being executed by the "right" address space. The Hook Service will allow a Z/XDC debugging session to be started only if it is being executed by the right address space.

Associating an address space with a hook can be done with the **ASID=** and **ASNAME=** operands of the HOOK command.

Unfortunately, when the Hook Service finds that the hook has been reached by execution in the **wrong** address space, it cannot just ignore the situation. It cannot simply allow the calling program to resume as if nothing happened. This is because for dynamic hooks, six bytes of the target program's code had to be removed (and saved elsewhere) in order to set the hook into the code. If the Hook Service tried to ignore that fact and let the program resume execution, those 6 bytes of code would not be executed, and so the program would fail in potentially wild and crazy ways. That, of course, is not acceptable.

Accordingly, when the Hook Service discovers that it is being executed by the wrong address space as the result of a dynamic hook, it has to do the following:

- It restores the original code to the hook location, thereby removing the dynamic hook from code.
- It adjusts the caller's PSW so that the restored code will be executed.
- It issues WTO messages to the System Log to alert you that this has happened. The messages are:

```
DBC527E xxx HOOK PROCESSING FAILURE - RC=hh, REASON=hhhhhhhh
DBC527E THE DEFINITIONS OF THESE CODES CAN BE FOUND WITHIN THE #XDCHOOK
        MACRO
DBC527E ADDITIONAL INFORMATION FOLLOWS
```

```
        DBC964E THE TARGET ADDRESS SPACE WAS NOT THE FIRST TO REACH THE HOOK
        DBC964E     TARGET ASPACE: asname1
        DBC964E     ACTUAL ASPACE: asname2
        DBC964E THE HOOK HAS BEEN ERASED (SO THAT "asname2" CAN PROCEED UNHINDERED)
```

  - It allows the caller to resume execution.

Thus, the calling program resumes as if nothing happened; however, the hook will no longer exist.

Note, this problem does not exist for static hooks, since program code did not have to be replaced in order to set the hook.

If for some reason the Hook Service finds that it cannot restore the calling program's code, then the calling program is ABENDed on the spot. (This is the safest thing to do since there is no way to predict what the caller would do if it were allowed to resume without being able to execute the missing 6 bytes of code.) The ABEND is via a DEAD trap; therefore, the ABEND code will be s0C1. Messages DBC965T are displayed to explain the reasons for the failure.

In summary, when the Hook Service is executed from common storage as the result of a dynamic hook, it behaves as follows:

  - It **always** tries to remove the hook and restore the caller's original code **regardless** of whether or not it is being executed from the "right" or "wrong" address space.
  - If the caller's code was successfully restored, then the caller's PSW is adjusted so that the restored code will be executed when the caller resumes execution.
  - If the Hook Service is being executed from the "right" address space, then the Service builds a Z/XDC debugging environment and then passes control to Z/XDC, thereby starting or resuming a debugging session.
  - Otherwise (if the Hook Service is being executed from the "wrong" address space), the Service will either allow the calling program to resume immediately, or it will ABEND, depending upon whether or not it was able to restore the caller's code.

When the HOOK command is used to set a hook into common storage, its default action is to use Z/XDC's current Foreign Address Space Mode to determine the address space that you want the hook to be targeted against. If Z/XDC is not running in Foreign Address Space Mode, then (by default) the hook will be targeted against the address space within which Z/XDC is running. See HELP VIRTMEM XDCACCESS FASM for more information.

BUT this default action can be overridden by the ASID= and ASNAME= by operands on the HOOK command. In fact, using the ASNAME= operand, you can even target an address space that does not yet exist. See HELP COMMANDS HOOK for more information.

Information about a hook, including the address space against which it is targeted, can be displayed via the **LIST HOOKS** command.

Help Hooks DYnamic Whathappens

When execution reaches a Dynamic Hook, a Z/XDC debugging session is either created or resumed and then immediately receives control. At this point, messages are displayed at your terminal, and you can now type in Z/XDC commands.

What gets displayed can appear confusing, so let me take a moment to explain what the Hook Service has to do.

What You Don't See

If all goes well, then most of the following events should be invisible.

- The **JLU** (i.e. the hook) jumps to the Hook Service Prolog. This, along with the **HOOKCBE** control block in which it is contained, is a private and unique instance of code and data controls and work area.

- The Prolog's job is to:
  - Save registers and state data.
  - Normalize to a known state.
  - Erase the hook by restoring the original code to the Hook Point.
  - Locate and jump to common Hook Services code (called **Phase 2** code).
- Phase 2 locates the **xxx** load module (usually the **XDC** load module) and then sets it up either as an **ESTAEX** or an **FRR** according to the hook'd program's execution environment.
- Phase 2 then briefly enters Z/XDC with instructions to set a transient trap at the Hook Point (so that Z/XDC will receive control again when the Hook Service eventually returns execution back to the Hook Point).
- If any error conditions have occurred, then error messages are issued to SYSLOG.
- Phase 2 then restores **all** registers and State Data and then returns to the Hook Point.
- When execution returns at the Hook Point, the transient trap that the Hook Service caused Z/XDC to plant there is executed causing Z/XDC to receive control again and issue displays.
- At this point, you can issue your debugging commands.

What You Do See

During this process, Z/XDC receives control **twice!** Each time causes messages to be displayed that, absent explanation, can be confusing. So here's what's happening:

- The first entry into Z/XDC is a brief one that arises from a **TYPE=CMD** type **DEAD trap.** that is issued by the Phase 2 portion of Hook Support. (Such DEAD traps can be used to pass command strings to Z/XDC for execution. See HELP BREAKPOINTS DEADTRAPS for more information.) This results in the following messages (more or less) being displayed.

```
DBC830I xxx release (maintenance level)
DBC830I ENTERED *state*, AMODE(nn), UNDER RB#n FROM TCB#n IN asname
        (ASN=asid.pin)
DBC891I xxx v3.1 AS AN estaeorfrr OWNED BY RB#n
DBC831I THE ERROR LEVEL AND RETRY LEVEL ENVIRONMENTS ARE THE SAME

DBC842I DEAD trap CMDS: EQUATE HOOKCBE RW13~INDIRECT(64) DATA;TRAP
        RW1~INDIRECT(64,ZEROOK) 'DELETE HOOKS HOOKCBE;OFF PSWE!;WHERE'
        SAVE=NO;GOT
```

  The **DBC842I** message reports the string of commands passed into Z/XDC by the DEAD

trap. They're pretty complicated, but basically they tell Z/XDC to place a transient breakpoint at the Hook Point and then return control back to the Phase 2 Hook Support code.

The command string associated with the breakpoint will not be executed until Hook Services returns execution to the Hook Point, thus causing that breakpoint to be executed, thus returning control to Z/XDC a second time.

- The second entry into Z/XDC occurs when Hook Services completes its work and returns execution to (the now restored) hook point where the transient breakpoint is waiting.

Z/XDC disables the breakpoint (because it is, after all, transient) and then processes the automatic commands that had been associated with the breakpoint. They basically just clean up the remnants of both the hook and the breakpoint, and then issue a **WHERE** command to display the Hook Point code.

Automatic Commands processing will produce the following messages:

```
 TRAP - opcode
>>> DELETE HOOKS HOOKCBE <<<
>>> OFF PSWE! <<<
>>> WHERE <<<
[Followed by the WHERE command's display]
```

Error Handling

It is important to note that Dynamic Hooks preserve all states and **all registers** across the processing that the Hook Service does. In particular (and unlike what happens for Static Hooks), this means that the Hook Service cannot return any return codes or reason codes. Instead, the Hook Service deals with errors according to whether they are **recoverable** errors or **NONrecoverable** errors.

But regardless of whether or not an error occurs, one of the very first things that the Hook Service does is it tries to erase the hook by restoring original code to the Hook Point. In fact, when errors occur, the single distinction between recoverable and nonrecoverable errors is whether or not the hook was successfully erased.

So, for **Recoverable Errors,** the Hook Service does the following:
- It erases the hook by restoring original code to the Hook Point.
- It issues error messages to SYSLOG.
- It does not create a debugging session.
- Instead, it just returns execution to the Hook Point so that the interrupted program can continue executing as if nothing happened.

For **NONrecoverable Errors,** the following happens:
- The Hook Service was **unable** to erase the hook, so the target code remains corrupted!
- Error messages are written to SYSLOG.
- The interrupted execution thread, being corrupted, cannot be allowed to continue.
- So the execution thread is **ABENDed** (with an s0C1) at a DEAD trap.
- The DEAD trap itself contains an error message containing additional information.

Help Hooks Static

A **static** hook is a hook that is not created via Z/XDC's HOOK or HDEFERRED command. Generally, a static hook is one which is created at program assembly time. It is a permanent part of the program and, therefore, not one that can be removed.

Coding a static Z/XDC hook into your program is functionally similar to coding a **LOAD-ESTAE-#DIE** sequence. When the hook is executed:

- A copy of Z/XDC is located into storage (like a LOAD macro does),
- System control blocks are created to make Z/XDC your program's newest ESTAE or FRR, (like an ESTAEX or SETFRR macro does).

- And control is breakpointed to Z/XDC (like a #DIE macro does).

One additional thing that a Z/XDC hook does is deal with (E)SPIEs. If the Hook Service finds that an (E)SPIE exists, it modifies the (E)SPIE to prevent it from intercepting program check ABENDs s0C1 thru s0C7. (The (E)SPIE's ability to intercept all other program check ABENDs, 0C8 thru 0CF, is not changed.) For more information, please see HELP DEBUGGING ESTAES SUPERSEDING.

Static hooks are created by the **#XDCHOOK** macro, which generates logic to locate the Hook Support routines and then BASR to them.

For C programmers, a static hook can be created by calling the **cxdchook** function. See HELP DEBUGGING C CXDCHOOK for more information.

In prior releases of Z/XDC, the #XDCHOOK macro used to find and EXecute an SVC routine to perform the service of creating a debugging session, but that is no longer the case. Now, the code generated by #XDCHOOK simply BASRs to Hook Services.

However, for Legacy Support reasons, Z/XDC still has a HOOK SVC routine. This is so that previously assembled code containing #XDCHOOK macros does not have to be reassembled. The old object code will still work.

But if you do reassemble old code, new logic will be generated.

Notice that the old hook SVC remains supported. This means four things:

- As just noted, old code assembled with the old version of #XDCHOOK will still work.
- The installed hook SVC number will continue to be displayed by the **LIST XDC** command.
- If (against recommendation) you still want to avoid assembling the #XDCHOOK macro, and instead assembled (or zap) a hook SVC instruction, directly into your code, you can still do that. Just be sure you are using the correct SVC number, and be aware that there are many circumstances under which it might not be SVC 199!

Static hooks cannot be displayed by the **LIST HOOKS** command. That command only displays dynamic hooks created by the HOOK command. However, if a location containing a static hook is displayed via a FORMAT command, the #XDCHOOK logic will, of course, show up.

Static hooks cannot be removed by the **DELETE HOOKS** command. That command only removes dynamic hooks (created by the HOOK command). However, the ZAP command can,

of course, be used pretty easily to defeat the #XDCHOOK's generated logic.

Subtopics Index

The following topics provide further information about static hooks. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

#XDCHOOK - Specific information for Assembler programmers about the #XDCHOOK macro.
cxdchook - C programmers can learn about the **cxdchook** subroutine at HELP DEBUGGING C CXDCHOOK.

Help Hooks Static #xdchook

If you want to assemble a static hook into your program, then you must use the **#XDCHOOK** macro to do so. This macro emits code that locates and calls Z/XDC's Hook Services routines.

The #XDCHOOK macro can be found in Z/XDC's **XDCMACS** library. The library's factory default name is hlq.XDCV31.XDCMACS. Ask your Systems Programmer for its actual name at your Data Center.

#XDCHOOK uses several internal macros: #REGS, #TEST, #DBCVRSN, #DBCHDAT and @SYSTATE. These macros can also be found in the **XDCMACS** library.

These inner macros **must** be at a compatible maintenance level. If they are not, then #XDCHOOK will decline to expand. (Of course, when #XDCHOOK is loaded from the same library as the inner macros, compatibility is pretty much guaranteed.)

#XDCHOOK Redesign and Legacy Support

In Z/XDC release z2.1, Hook Processing was entirely redesigned so as to discontinue implementing Hook Processing via an SVC instruction. So now neither Dynamic Hooks nor Static Hooks use an SVC instruction anymore. Instead:

- Dynamic Hooks use a **JLU** to jump to the start of Hook Processing.
- Static Hooks use **BASSM** (or something similar).

The purpose, of course, is to allow Hooks to be set into SVC unfriendly code.

One consequence of all this is that the #XDCHOOK macro, starting with release z2.1, is significantly different from prior versions. However, **both** of the following statements remain true:

- Older versions of the #XDCHOOK macro can still be used with newer releases of Z/XDC. This is because a legacy copy of the Hook SVC is still a part of the product.

- Newer versions of the #XDCHOOK macro can be used with older releases of Z/XDC. This is because the macro's emitted code detects at run time the level of Z/XDC installed and issues the Hook SVC when an older level is detected.

Consequently:

- Existing code that was previously assembled with the older #XDCHOOK macro does not have to be reassembled.

- When existing code is reassembled, the older #XDCHOOK logic will automatically be replaced by the newer logic.

- Code assembled with the newer #XDCHOOK logic will still function correctly when taken to a System on which an older release of Z/XDC is installed.

All macro operands supported by the older #XDCHOOK macro...
- Either are retained with their same meanings in the newer version,
- Or are obsolete but are still present so as to avoid assembly errors.

The newer version of the macro supports several newer operands to deal with new conditions that arise from Hook Processing's expanded usability.

Macro Syntax

The following is a simplified description of #XDCHOOK's syntax. This description should satisfy the needs of most users; however, for those who are curious, a comprehensive description can be found in header commentary located in the macro's source code in the XDCMACS library.

Syntax:

```
    name #XDCHOOK XDC=xxx,
                  ERROR=address,
                  TRAPAT=address,

                  FINDHOOKIFACE=YES,
                  FINDHOOKIFACE=NO, [the default]
                  HOOKIFACE=,

                  MAPS=YES, [the default]
                  MAPS=NO,

                  VCHECKS=PARANOID, [the default]
                  VCHECKS=RISKY,

                  RCEQUATES=(YES,ONLY)
```

All operands are optional.

name
This allows you to assign a statement label to the code emitted by the macro. If this is omitted, then no label is assigned.

XDC=xxx

This allows you to specify the name of the Z/XDC clone that your program is going to be debugged with. It is possible that Z/XDC has been installed on your computer with a 3-character name other than "XDC". (That would be called a "clone name".) If that is the case, then you must use this operand to provide that name so that the code emitted by #XDCHOOK searches the right Z/XDC control block structure. If this operand is omitted, then #XDCHOOK assumes that Z/XDC has not been renamed. If Z/XDC has been renamed, then you should be able to find out from your Systems Programmer what that name is.

XDC= may be given in the following ways:

XDC='xxx'
A string that is exactly 3 characters long. The characters must form a syntactically valid load module name. Example:

```
            #XDCHOOK XDC='V31'
```

XDC=(reg)
The name of any general register, in which case the Z/XDC clone name must have been preloaded into the 3 hi-order bytes of the register's lo-order word (i.e. into the 5th, 6th and 7th bytes of the 8-byte wide register). Example:

```
            ICM   R5,B'1110',=C'V31'
            #XDCHOOK XDC=(R5)
```

XDC=address
An "rx-type" address expression that points to the location of a 3-character clone name. Examples:

```
            #XDCHOOK XDC=XDCNAME
            ...
   XDCNAME  DC    C'V31'
```

or:

```
            #XDCHOOK XDC==C'V31'   ("==" is not a typo.)
            ...
            LTORG ,
            =C'V31'
```

For more information about renaming Z/XDC, see **HELP XDCCLONES.**

ERROR=address
It is possible that the emitted code will fail to find Z/XDC's Hook Service. If the Service is found and processes successfully, then execution will "fall through" to the next statement following the macro. However, if the Service is not found or if it fails, and if this **ERROR=address** operand is given, then execution will jump to the given address.

If **ERROR=address** is **not** given, then the emitted code will fall through regardless of whether or not the Service is found and ran successfully. In this case, return and reason codes will have to be checked.

ERROR= may be given in the following ways:

ERROR=address
An "rx-type" address expression that points to a routine that is to receive control if an error occurred. Example:

```
            #XDCHOOK ERROR=HOOKFAIL
```

```
            ...
   HOOKFAIL [error handling logic]
            ...
```

ERROR=(reg)
reg may be any general register other that R0 and R1. Also, it may not conflict with other registers that might be used by the code emitted by the macro. (Exactly which registers those might be depends upon the operands that you give to the macro.) You need to preload the address of the error routine into the register before issuing the macro. Example:

```
            LA    R5,HOOKFAIL
            #XDCHOOK ERROR=(R5)
```

TRAPAT=address
Normally, Hook Services sets a trap at the **macro's return point** so that Z/XDC receives control once the hook has been processed and a debugging environment created. You can use the **TRAPAT=address** operand to change that first trap location to somewhere else. That way, when Hook Services finishes creating the debugging environment and returns to its caller (your code), Z/XDC will **not** immediately receive control. Instead, your program will continue running until it reaches the location provided by the **TRAPAT=** operand.

TRAPAT= may be given in the following ways:

TRAPAT=address
An "rx-type" address expression that points to where the first trap is to be set. Example:

```
            #XDCHOOK TRAPAT=STOPHERE
            ...
   STOPHERE [first trap location]
            ...
```

TRAPAT=(reg)
reg may be any general register other that R0 and R1. Also, it may not conflict with other registers that might be used by the code emitted by the macro. (Exactly which registers those might be depends upon the operands that you give to the macro.) You need to preload into the register the address of the place where you want your first trap location to be. Example:

```
            LA    R5,STOPHERE
            #XDCHOOK TRAPAT=(R5)
```

FINDHOOKIFACE=yesorno
HOOKIFACE=address
When the #XDCHOOK macro is located in locked or disabled code, the only storage that its logic is allowed to touch must be either DREF storage or page fixed storage. Normally, the emitted logic finds Z/XDC's Hook Services via a lookup against ColeSoft's **Customer Anchor Table.** This lookup logic meets the requirement of touching only DREF or page fixed storage. Therefore, it is safe to use this form of the macro in locked or disabled code.

On the other hand, the instance of Z/XDC named **XDC** is the **only** clone that is connected to our Customer Anchor Table. So if you are using **#XDCHOOK** with any **other clone** of XDC (i.e. by using the **XDC=** operand), the macro cannot find that clone's

Hook Services via the Anchor Table. Instead, the macro has to emit logic that finds Hook Services by scanning the System's SubSystem Control Tables (SSCTs). But those are located in pageable storage, so they cannot be scanned while disabled.

The **FINDHOOKIFACE=** and **HOOKIFACE=** operands provide a way around this problem.

FINDHOOKIFACE=YES
This operand causes #XDCHOOK to emit logic to find the Hook Services routine but **not** to call it. Instead, the routine's address is returned in R0, which you can then squirrel away (presumedly in a nonpageable place) for later use.

You would use **FINDHOOKIFACE=YES** in some part of your program that runs enabled.

Notes:
- The Hook Services routine will always be located in 24-bit page-fixed storage.
- The location where you save the returned Hook Services address probably needs to be in either DREF or page fixed storage.

HOOKIFACE=address
You would use this operand when you want to place the #XDCHOOK macro into locked code. For **=address,** you would provide the address previously obtained though the FINDHOOKIFACE=YES call.

HOOKIFACE= may be given in the following ways:

HOOKIFACE=address
This is an RX-type address expression that points to the field into which you previously stashed the Hook Services address. Important! This field must be located in page fixed or DREF storage.

HOOKIFACE=(reg)
This is the name of a register into which you have loaded the address of the Hook Services routine. All hi-order bits should be zeroed.

MAPS=YES [the default]
MAPS=NO
The z/OS system control blocks that are potentially referenced by the emitted code are the CVT, the ECVT, IBM's Customer anchor Table, JESCT, and the SSCTs.

The Z/XDC control blocks potentially referenced by the emitted code are the CSWANCHR and DBCCVT.

The **MAPS=** operand allows you to control whether the emitted code will refer to these control blocks via field names (MAPS=YES or omitted) or via magic number hexadecimal offsets (MAPS=NO).

The default is **MAPS=YES.** But if a needed map has not been provided, the macro will still use magic number offsets for the unavailable symbols so as not to create assembly errors. Example:

```
            CVT   DSECT=YES
   ******** IEFJESCT , [Omit this map, but that will be ok]

            IEFJSCVT ,
```

```
            ...
            #XDCHOOK MAPS=YES  [or MAPS= omitted]
```

Code **MAPS=NO** if, for some reason, you want the macro not to use field names even when they are available.

VCHECKS=PARANOID, [the default]
VCHECKS=RISKY,
As one customer remarked recently, the code emitted by this macro "has a rather large footprint". That's mainly because the macro emits numerous validity checks. You can use this operand to reduce that.

It is exceedingly rare that a vcheck ever actually detects anything, so you might be tempted to remove them. But if you do so, it will be at your own risk!

VCHECKS=PARANOID [or omitted]
The macro expansion includes all validity checks. This is the recommended setting.

VCHECKS=RISKY
The macro expansion emits almost no validity checks. It presumes that all z/XDC related control blocks are installed and current.

This is a risky setting, but as the Pope said a few years ago, who am I to judge.

RCEQUATES=(YES,ONLY)
This causes the #XDCHOOK macro to emit a complete set of equates to represent all possible return codes and reason codes that can occur in Hook Processing (some for static hooks, some for dynamic, some for both).

The **ONLY** suboperand tells the macro to only emit the equates and not to also emit static hook logic.

When the **RCEQUATES=** operand is not given, then the equates will be automatically emitted on the first use of the #XDCHOOK macro that does not specify RCEQUATES=NO.

Other Macro Operands

The #XDCHOOK macro also accepts the following additional operands:
- **LIST=** For controlling whether or not the macro expansion is to be displayed.
- **RELEASE=** For identifying a particular release of Z/XDC.
- **RELIDC=** Another way to identify a particular release of Z/XDC.
- **PFIX=** For identifying a particular register set to be used by the emitted code.

Usually, these operands would not be needed, but if you want to find out more about them, then you can read the commentary found in the macro's source code in the **XDCMACS** library.

Finally, the following operands are obsolete. They remain present so as to avoid unnecessary assembly errors, but they are ignored.:
- AMODE64= The emitted code is now AMODE agnostic.
- RI= The emitted code now always uses relative immediate machine

instruction.

Permissible States

#XDCHOOK can be used in code running in any of the following states:

- AMODE: Any (24-bit, 31-bit or 64-bit)
- Execution Mode: Any (TCB or SRB)
- ASC-MODE: Almost any (just not HOME)
- Cross Memory: Any (HASID=<>PASID=<>SASID)
- State: Any (Supervisor or problem)
- Key: Any (0 thru 15)
- Enablement: Any (Enabled or disabled)
- Locks: Any
- Registers: Any
- Linkage Stack: Slots must be available
- Recovery environment: Doesn’t matter

States Upon Return

- AMODE: restored
- Execution Mode: unchanged
- ASC-MODE: restored
- Cross Memory: restored
- State: restored
- Key: restored
- Enablement: unchanged
- Locks: have been released and then restored (See **HELP DEBUGGING LOSTLOCKS** for more information)
- Linkage Stack: restored
- Recovery environment: z/XDC is now the newest recovery routine (ESTAEX or FRR)

Register Usage

- R15: Return code
- R0: Reason code

- RH15: Additional information or =0
- RH0: Additional information or =0

- R14 RH14 R1 and RH1 may be destroyed.
- All other registers are restored.

Return and Reason Codes

Use **#XDCHOOK RCEQUATES=(YES,ONLY)** (and no other operands) to emit a complete set of return code and reason code EQUs. D

Help Hooks Static Cxdchook

Move along now, move along. Nothing to see here, move along.

What you're looking for can be found over at HELP DEBUGGING C CXDCHOOK. Move along now.

Help Hooks DEferred

Z/XDC's dynamic hooks can be either "active" or "deferred". An active hook is one that has been set into a program that is currently resident in storage. A deferred hook is one that is not yet set. Deferred hooks are targeted towards load modules that are not yet resident in storage. They will not be set until the targeted load module is loaded into private storage by z/OS

Z/XDC also supports deferred breakpoints. Like deferred hooks, these are breakpoints that are targeted towards load modules (or program objects) that will be brought into storage in the future. For more information about deferred breakpoints, see HELP BREAKPOINTS DEFERRED.

When a deferred hook is created, then at the moment that a targeted module is brought into storage, an active copy of the deferred hook is set into the module at the designated offset. All of the deferred hook's attributes are copied when creating the active hook.

Deferred hooks cannot target modules to be loaded into common storage. They can target only modules that are to be loaded into private storage.

Deferred hooks cannot be used to violate security. When a targeted module is loaded into storage, Z/XDC checks to ensure that the module's storage key matches the execution key of the user program at the time that the HDEFERRED command was issued. Thus, for example, a problem mode program can successfully target only modules that are loaded into key 8 or key 9 storage.

Issues Regarding Reentrant and Refreshable Load Modules

Please note that reentrant programs from authorized libraries are normally loaded into key 0 storage. Thus, if a problem mode program attempts to target a deferred hook towards such a module, then the attempt will fail. ***BUT*** there is a pretty easy way to tell the System to load rentrant programs into key 8 storage (instead of key 0). See HELP DEBUGGING REENTRANT for more information.

Similarly, on systems that have the **REFRPROT** facility both installed and activated, programs that are refreshable (REFR) are (like rentrant programs) also loaded by the System into key 0 storage (regardless of the load library's authorization level and regardless of the RENT attribute). Therefore, when Z/XDC is running non-authorized, hooks cannot be placed in such programs without first turning REFRPROT off. But again, there is a pretty easy way to locally turn REFRPROT off. See HELP DEBUGGING REFRESHABLE for more information.

Previously Loaded Rentrant and Refreshable Load Modules

A deferred hook **never** generates an active hook into copies of a target module that are already resident in storage. This means:

- If a copy of a target module is already in storage at the time that an HDEFERRED command is issued, the resident copy remains unaffected. If you need to place an active hook into that copy, then you will have to use the HOOK command.

- If a target module has the **RENT** or **REFR** Binder attributes, then the System may satisfy LOAD, LINK, ATTACH, and XCTL requests by returning a pointer to a copy of the target module that is already in storage due to a previous request. Thus, the System may decide to avoid reading a new copy of the module into storage. When this occurs, Z/XDC's code that clones active hooks from deferred ones is bypassed, and so cloning does not occur. Cloning occurs only when a load module is actually placed into private storage.

Hook Placement Within a Load Module

A deferred hook can be targeted towards:
- A load module's (or program object's) entry point.
- Its load point.
- The start of any csect within the module.
- Or any offset relative to any of the above, as long as the resulting location falls within the actual limits of the load module or program object.

To refer to a location that is relative to the module's entry point, just specify the module's name, plus or minus an offset. Examples:

```
HD MYMODULE
HD MYMODULE+3DC
HD MYMODULE-1C   - (valid if the module's entry point is not located at its load
                   point)
```

If a load module's entry point is not located at its load point, and if you want to target a deferred hook relative to the physical start instead of to the entry point, then just use Z/XDC's **.X#1** syntax for referring to a load module's first extent. Examples:

```
HD MYMODULE.X#1
HD MYMODULE.X#1+35C
```

Placing a Hook Relative to a Csect Within a Load Module

If you wish to target a deferred hook relative to a csect within a load module, then you must first make the module's Binder map available to Z/XDC so that it has a way to determine the csect locations within the module. To do this you must use Z/XDC's **DMAP** command to read the module's Binder map as if it were a dsect. Example:

```
DMAP MYMODULE.
```

The trailing dot (.) is important! It is the presence of this dot that informs Z/XDC that you want the DMAP command to read a Binder map.

Once you have read the Binder map, you can then reference the load module's csects

with the HDEFERRED command. You do **not** need to issue the USING command to place the map anywhere. It is sufficient that the map has been read, it does not need to be assigned to storage. Examples:

```
DM MYMODULE.
HD MYMODULE.MYCSECT
HD MYMODULE.MYCSECT+A0
HD MYMODULE.MYCSECT-3C
```

Deferred hooks in overlay modules are not supported anywhere but in the root segment.

Deferred hooks are created by the HDEFERRED command. Active hooks are created by the HOOK command. For more information, please see the following:

```
HELP COMMANDS HOOK
HELP COMMANDS HDEFERRED
```

Help Hooks Legacysupport

In prior releases of Z/XDC, the Hook Service was implemented via an SVC routine (usually SVC 199 but not always). That is no longer the case. (See HELP WHATSNEW Z21 HOOKREWRITE for more information.)

Now, the Hook Service is **branch entered** directly. This is so that hooks can be placed in such **SVC-unfriendly** code as PC routines, SRB routines, FRR protected routines and the like.

Accordingly, every aspect of Hook Support Services has changed, but in particular, the **#XDCHOOK** macro has been rewritten. Previously, it would search Z/XDC control blocks to find a standard HOOK SVC instruction, and it would EXecute that. Now, however, it searches for the Hook Service's prolog code and then simply **BASRs** to it. (See HELP HOOKS STATIC for more information.)

Nevertheless, for **legacy support** reasons, z/XDC still installs a **HOOK SVC.** This is because previously assembled code containing #XDCHOOK macros still contains logic that uses the HOOK SVC.

When any code that contains a #XDCHOOK macro is reassembled, the generated logic will change significantly. We do not expect that will create any issues, but if it does, then please let us know.

Help Hooks Undoing

Hooks are created by either:

- The HOOK command, or
- The **HDEFERRED** command, or
- The **#XDCHOOK** macro.

In either case, when the hook is reached by execution, an **ESTAEX** or an **FRR** is create

with Z/XDC as the recovery routine. This makes Z/XDC the **newest** recovery routine at that point in your code, and so when a breakpoint (or DEAD trap or other ABEND) is executed in that code, Z/XDC is the first recovery routine that gets to see it.

This is all well and good as far as it goes. But often a hook is necessary because user code has already established **its own** ESTAE[X] (or some such), and in order for Z/XDC to work, it has to get in front of that! (Which is what the Hook Service does.)

But the **unexpected** appearance (from the program's point of view) of this hook-created ESTAEX/FRR can lead to serious problems! In the event that the user program eventually tries to delete its own recovery routine, it will instead delete the hook-created ESTAEX/FRR, and then all hell will break loose! This is because the user program will proceed as if its recovery routine is gone when in fact it is still present. This can lead to **arbitrary failures.**

The solution to this problem is this: When you are done with debugging your routine and you want to let it just continue on its merry way, then issue **GO REMOVEXDC.** This command causes your program to resume execution, but while doing so, the ESTAEX or FRR which was created by the hook (and by which Z/XDC had been receiving control) is removed from the recovery environment. So it will no longer interfere with your code's own recovery cleanup logic.

There is more detailed information available at HELP COMMANDS GO.

Help DDnames

There are a large number of ddnames that are recognized and acted upon by various parts of the Z/XDC product, and they serve a wide variety of purposes, not all of them having to do with datasets.

This topic is a central point where each and every ddname used by Z/XDC is described. But before I start listing them all, I first need to explain a couple of, perhaps unusual, things about them.

DDnames and Clone Names

One rather unique feature of Z/XDC's is the concept of an **XDC clone.** This is the ability to create two complete and entirely separate instances of Z/XDC simply by running through its load libraries and renaming the first three characters of its load module names from **XDC** to something else. (Throughout Z/XDC's Built-in Help, that "something else" is represented as **xxx.)** Doing this affects nearly **all** of Z/XDC's references to every label, name and string that start with the characters **XDC,** including ddnames.

For more information about Z/XDC clone names, see **HELP XDCCLONES.** Meanwhile, at this point, I'll confine myself to discussing the affect of clonenames on ddnames.

Most of the ddnames recognized by Z/XDC start with Z/XDC's clone name. Normally, that clone name is **XDC;** however, when Z/XDC has been renamed to some other 3-character name, Z/XDC will look for ddnames starting with those three characters

instead of **XDC.** So in this topic, I'll represent such ddnames with **xxx** substituted for **XDC.**

Keyword DDnames

Not all of the ddnames recognized by Z/XDC are for the purpose of pointing to datasets. There are a large number of ddnames that Z/XDC looks for but never OPENs. These serve the purpose of communicating control information into the debugging session.

These ddnames are called **keyword ddnames,** and Z/XDC will either take or not take a particular action simply by virtue of their presence or absence.

All of the keyword ddnames (as well as most of the dataset-linked ddnames) start with Z/XDC's clone name. That way, when two or more clones of Z/XDC are running in the same environment, the clones won't get confused over which ddname is intended for which clone.

Z/XDC Recognized DDnames

For convenience, I'll represent the ddnames with leading slashes (as if from JCL), but of course in TSO you'd use the ALLOCATE command, instead of JCL, to create these ddnames. Examples:

```
JCL: //XDCRENT8 DD DUMMY
TSO: ALLOCATE DDNAME(XDCRENT8) DUMMY REUSE
```

The following ddnames are recognized by or are relevant to Z/XDC and its various support programs.

For detailed information, type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

ADATA - **//xxxADATA DD DSN=sequentialfile** - An ADATA output file from the ADATA Filtering Program (named **xxxADATA).**

ADUMP - **//xxxADUMP DD DSN=sequentialfile** - A diagnostic output file for the ADATA Filtering Program **(xxxADATA).**

CDF - **//xxxCDF DD DUMMY** - A keyword ddname directing Z/XDC to communicate with the user via cdf/XDC.

CDFTRACE - **//CDFTRACE DD SYSOUT=class** - A dynamically allocated file used by cdf/XDC for a diagnostic trace.

CMDS - **//xxxCMDS DD DSN=sequentialfile** - A script of Z/XDC commands to be automatically executed at the start of a debugging session.

COFF - **//xxxCOFF DD DUMMY** - A keyword ddname that directs Z/XDC to suppress all commands and features that make up c/XDC.

FPTnn - **//xxxFPTnn DD DUMMY** - A control for FRR Proxy Task allocations

(Notice: **FPT** not FTP).

ITRAC - **//xxxITRAC DD DSN=sequentialfile** - The default ddname used by the SET XYZZY ITRACE EXPORT command when exporting the contents of the ITRACE buffer out to DASD.

JSTCB - **//xxxJSTCB DD DUMMY** - A control affecting whether or not xxxCALLA is to ATTACH the target program as a **jobstep** task.

JSTEP - **//xxxJSTEP DD DUMMY** - A control affecting the multitasking scope of a debugging session (should **always** be used in batch).

LIB - **//xxxLIB DD DSN=loadlibraries** - Z/XDC's load library (when not in a System library or a STEPLIB).

MAPLB - **//xxxMAPLB DD DSN=adatalibraries** - Libraries or files containing ADATA for use by the MAP command.

NCOMP - **//xxxNCOMP DD DUMMY** - A keyword ddname (used by the xxxADATA program) that suppresses Z/XDC ADATA compression.

NOALC - **//xxxNOALC DD DUMMY** - A keyword ddname that directs Z/XDC to suppress all commands that make use of dynamic allocation.

NOMAP - **//xxxNOMAP DD DUMMY** - A keyword ddname that suppresses **AUTOMAPing.**

NOSC - **//xxxNOSC DD DUMMY** - A keyword ddname that controls whether or not dump/XDC compresses the journaled session state data that it preserves when a dump/XDC session is suspended. (dump/XDC only).

NSTEP - **//xxxNSTEP DD DUMMY** - A keyword ddname that suppresses **AUTOSTEP'ing.**

PLIB - **//xxxPLIB DD DSN=panellibrary**
- //ISPPLIB DD DSN=panellibrary - An ISPF-type panel library needed by Z/XDC.

PRINT - **//xxxPRINT DD SYSOUT=class** - The output file used by Z/XDC's **PRINT** command. Also a report file created by the ADATA Filtering Program (named **xxxADATA).**

PROF - **//xxxPROF DD DSN=profilelibrary**
- //ISPPROF DD DSN=profilelibrary - An FB-80, ISPF-type profile library for storing Z/XDC profile files.

QUICK - **//xxxQUICK DD DUMMY** - A keyword ddname directing xxxCALL[A] to forego the initial Z/XDC display and allow the user program to start immediately.

REFR8 - **//xxxREFR8 DD DUMMY** - A keyword ddname directing xxxCALL[A] to cause refreshable programs to be loaded into key 8 storage.

RENT8 - **//xxxRENT8 DD DUMMY** - A keyword ddname directing xxxCALL[A] to cause reentrant programs to be loaded into key 8 storage.

REXX - **//xxxREXX DD DSN=rexxlibraries** - Libraries of REXX programs that can be executed as Z/XDC commands.

SDUMP - **//xxxSDUMP DD DUMMY** - A keyword ddname directing Z/XDC to create a System

Dump if it experiences a failure.

XCLUD - **//xxxXCLUD DD DSN=excludefile** - An input file (used by the xxxADATA program) containing an exclusion list.

SYSADATA - **//SYSADATA DD DSN=adatafile** - An output file used by the High Level Assembler (ASMA90) to write ADATA.

SYSCDBG - **//SYSCDBG DD DSN=dwarfdatafile** - An output file used by the XL C/C++ and Metal C compiler to write Dwarf Data.

SYSMDUMP - **//SYSMDUMP DD DSN=sequentialfile** - An output file to which the System will write a binary storage dump when an ABEND occurs that is not recovered. (You can then examine the dump using **dump/XDC.)**

SYS000nn - **//SYS000nn DD SYSOUT=class**
- **//SYS000nn DD DSN=sequentialfile** - A dynamically allocated output file containing a copy of the debugging session's commands and displays log.

TASKLIB - **//TASKLIB DD DSN=loadlibraries**
- **//xxxT0006 DD DSN=loadlibraries** - A set of load libraries that **xxxCALL[A]** sets up as Task Libraries for the programs to be debugged.

TLIB - **//xxxTLIB DD DSN=panellibrary**
- **//ISPTUSR DD DSN=panellibrary**
- **//ISPTLIB DD DSN=tablelibrary** - An ISPF-type table library needed by Z/XDC.

TRACE - **//xxxTRACE DD SYSOUT=class** - This ddname causes Z/XDC to write to SYSOUT a diagnostic trace of Z/XDC's terminal communications traffic.

TRSAF - **//xxxTRSAF DD DUMMY** - A keyword ddname directing Z/XDC to activate security call tracing immediately upon the start of a debugging session.

TSPRT - **//xxxTSPRT DD SYSOUT=class**
- **//SYSTSPRT DD SYSOUT=class** - The output file used by REXX's **SAY** instruction.

WTOR - **//xxxWTOR DD DUMMY** - A keyword ddname directing Z/XDC to communicate with the user via WTOs and WTORs.

XDCIS - **//XDCISxxx DD DUMMY** - A keyword ddname to tell a user program what XDC's current name is.

Help DDnames ADAta

//xxxADATA DD DSN=sequentialfile

This ddname is used only by the **xxxADATA Filtering Program.** This is a sequential output file to which filtered ADATA is written. For more information, see HELP MAPS ADATA EXCESSADATA XDCADATA.

Help DDnames ADUmp

//xxxADUMP DD DSN=sequentialfile

This ddname is used only by the **xxxADATA Filtering Program.** This is a sequential output file to which a lengthy, detailed processing report is written. For more information, see HELP MAPS ADATA EXCESSADATA XDCADATA.

Help DDnames CDF

//xxxCDF DD DUMMY
This a keyword ddname that affects the communications interface that Z/XDC may attempt to use when looking for a way to talk to a programmer:

//xxxCDF DD DUMMY tells Z/XDC to attempt to use cdf/XDC. In the batch, this ddname usually is unnecessary because Z/XDC will attempt to use cdf/XDC by default.

However, in **TSO** this allocation has a very useful effect: If you need to debug a TSO program that itself uses the TSO terminal, then its usage and Z/XDC's usage can interfere with each other in very obstructive ways. But if you add an **xxxCDF** allocation, that tells Z/XDC **not** to use the user's terminal at all, but rather to open a cdf/XDC connection instead. Doing this avoids the interference.

For more information, see **HELP XDCSRVER CDF COMMUNICATIONS.**

Note, when **dump/XDC** is active, **//xxxCDF DD DUMMY** is ignored.

Help DDnames CDFTrace

CDFTRACE file
When a **MODIFY xxxCDF,TRACE ALL ON** System Operator command is issued, cdf/XDC allocates a spool file to ddname CDFTRACE and begins writing an internal terminal communications trace to it (similar to the //xxxTRACE file described elsewhere). This is a low level trace of all of the actions that Z/XDC takes to connect to a user's terminal, send displays to it, and receive commands and other fullscreen actions back.

The CDFTRACE ddname is dynamically allocated by cdf/XDC, but it can be preallocated by server/XDC's JCL.

For more information, see HELP MESSAGES DBC676.

Help DDnames CMds

//xxxCMDS DD DSN=sequentialfile

This ddname points to a sequential file (or PDS member) that contains a script of Z/XDC commands to be executed at the start of a debugging session. For more information, see HELP XDCCALL DDNAMES.

The commands script will be executed the first time Z/XDC proper receives and accepts control, ie. the first time Z/XDC is about to produce a display. This means that **//xxxCMDS** plays nice with **//xxxQUICK** (as follows):

- If **//xxxQUICK** is also present, then the //xxxCMDS script execution is delayed until the first ABEND occurs or the first **#DIE** or assembled hook is reached.
- If //xxxQUICK is **absent,** then Z/XDC proper will receive control before the program starts, so the commands script will be executed at that time.

This ddname will be recognized only when a debugging session is started by the xxxCALL utility. This utility has four names: xxxCALL, xxxCALLA, xxxCMD and xxxCMDA. This utility can be invoked by any of its names. It can be invoked directly either by JCL or as a TSO command. It also is invoked for debugging sessions started by options 1 or 2 if Z/XDC's Startup Panel in ISPF. For more information, see HELP XDCCALL.

This ddname can be automatically allocated by Z/XDC's Startup Panel in ISPF. Type HELP DEBUGGING ISPF PANEL for more information.

Help DDnames COff

//xxxCOFF DD DUMMY

When the //xxxCOFF ddname is present in the debugging session, the initial **SET CXDC** setting from the Session Profile is ignored, and the Session starts with SET CXDC **OFF** in effect. This directs Z/XDC to suppress all commands and features that make up c/XDC. See HELP COMMANDS SET CXDC for more information.

Help DDnames Fptnn

//xxxFPTnn DD DUMMY (FPT not FTP)

This ddname gives the number of Formal FRR Proxy Tasks that xxxCALLA is supposed to create. **nn** must be a 2-digit decimal number specifying the number of tasks. Its value may range from 00 to 99. When this ddname is omitted, xxxCALLA, by default, creates two Formal Proxy Tasks.

When Z/XDC is running as an FRR (for debugging SRBs and FRR protected tasks), it must use a **Proxy Task** for doing such things as communicating with the user and processing its commands. For more information, see HELP DEBUGGING FRR.

This ddname will be recognized only when a debugging session is started by the xxxCALLA utility.

If your program does not use xxxCALLA to create the debugging session, and if you wish to debug SRB routines and/or FRR-protected tasks, then you can use the **SET PROXYTASKS** to create the needed Proxy Tasks.

Help DDnames Itrac

//xxxITRAC DD DSN=sequentialfile
This ddname is the default ddname used by the SET XYZZY ITRACE EXPORT command when exporting the contents of the ITRACE buffer out to DASD. This facility exists for internal diagnostic purposes only and is not further documented within Built-in Help.

Help DDnames JSTCb

//xxxJSTCB DD DUMMY
This is a keyword ddname. When present, it signals xxxCALLA that it should ATTACH the target program as a **jobstep task.** When absent, the program is ATTACH'd as a normal task. For more information, see HELP XDCCALL JOBSTEPTASKS.

Help DDnames JSTEp

//xxxJSTEP DD DUMMY

This is a keyword ddname. It affects Z/XDC in its process for deciding what the **multitasking scope** of the debugging session is within an address space.

A debugging session's "scope" is the set of tasks that are considered to be a part of the debugging session, and the highest task in that set is considered to be the owner of the session.

When the **//xxxJSTEP** ddname is present, the session's owning task is forced to be the address space's **jobstep task** (pointed to by ASCBXTCB).

When //xxxJSTEP ddname is **absent,** Z/XDC evaluates a more complex set of rules. For more information, see the series of topics starting with **HELP MULTITASK.**

Generally:
- When debugging programs running within TSO, this ddname should **not** be used.
- But when debugging multitasking programs running in the batch, then this ddname **should** be used.

Help DDnames Lib

//xxxLIB DD DSN=loadlibraries
This ddname points to a load library that contains a particular copy of the xxx load modules to be executed. For more information, see HELP XDCCALL DDNAMES.

Normally, //xxxLIB is not needed because Z/XDC will be installed into System libraries (linklist and LPALIB).

This ddname will be recognized only when a debugging session is started by the xxxCALL utility. For more information, see HELP XDCCALL.

Help DDnames Maplb

//xxxMAPLB DD DSN=adatalibraries
//xxxMAPLB DD DSN=adatafiles
This is a concatenation of one or more libraries or sequential files (but not a mixture) containing ADATA for use by the MAP and DMAP commands. For more information, see HELP MAPS ADATA MAPLIBS.

Help DDnames NComp

//XDCNCOMP DD DUMMY
When the //XDCNCOMP ddname is present in a job step or TSO session that executes the XDCADATA utility, the Z/XDC output file (if present) is not compressed.

Normally, Z/XDC ADATA compression is enabled. However, if you have ADATA consumers other than Z/XDC, then you will want to use this ddname to suppress compression.

For more information, see **HELP UTILITIES XDCADATA.**

Help DDnames NOAlc

//xxxNOALC DD DUMMY
When the //xxxNOALC ddname is present in the debugging session, Z/XDC is prohibited from using dynamic allocation (SVC 99).

This affects all Z/XDC commands that interact with datasets, such as loading maps, setting maplibs, saving profiles, session logging, etc.

The purpose of //xxxNOALOC is to help those customers who need to debug products that have hooks or exits within z/OS’s Dynamic Allocation.

Help DDnames NOMap

//xxxNOMAP DD DUMMY
When the //xxxNOMAP ddname is present in the debugging session, the initial **SET AUTOMAP** setting from the Session Profile is ignored, and the Session starts with SET AUTOMAP **OFF** in effect. This prevents Z/XDC from automatically loading csect maps whenever execution reaches an unmapped module. See HELP COMMANDS SET AUTOMAP for more information.

Help DDnames NOSc

//xxxNOSC DD DUMMY
This ddname pertains to dump/XDC's internal process for saving journaled information into IPCS literals for preservation when a dump/XDC session is suspended and resumed later. This ddname controls whether or not that data is compressed before saving.

- When **//xxxNOSC DD DUMMY** is present, the saved data **is not** compressed.
- When **//xxxNOSC** is absent, the data **is** compressed, thus requiring fewer IPCS literals to hold it all.

The purpose of this ddname is to provide a fallback should problems manifest themselves in the compression/decompression process.

For more information about journaling, see **HELP DUMPS MAPPINGDATA JOURNALING.**

Help DDnames NStep

//xxxNSTEP DD DUMMY
When the //xxxNSTEP ddname is present in the debugging session, the initial **SET STEP AUTOSTEP=** setting from the Session Profile is ignored, and the Session starts with **SET STEP AUTOSTEP=OFF** in effect.

When you're debugging an XL C/C++ or Metal C program, and c/XDC is installed, and **AUTOSTEP=ON** is in effect at the start of the debugging session, execution is automatically advanced from the C program's entry address to the machine code backing the first language statement.

However, if you actually do want to use Z/XDC to step through all of that prolog code, adding this DD card will give you that opportunity.

Help DDnames PLib

//xxxPLIB DD DSN=panellibrary
//ISPPLIB DD DSN=panellibrary

//xxxTLIB DD DSN=tablelibrary
//ISPTUSR DD DSN=tablelibrary
//ISPTLIB DD DSN=tablelibrary
When Z/XDC is using ISPF Display Services to paint its terminal displays, when these ddnames are present, then Z/XDC will search these first (in the order listed) for the template panel definitions (xxxV31A and xxxV31B) that it uses for building its displays.

When the //xxx--- ddnames are absent, Z/XDC will still search the //ISP--- ddnames. For more information, see HELP FULLSCREEN ISPF.

These ddnames can be provided regardless of whether the debugging session is running within TSO or within the batch.

Help DDnames PRInt

//XDCPRINT DD SYSOUT=class
This is a SYSPRINT like output file that is used in two, unrelated places:

- **Within Z/XDC itself,** this file can be preallocated by you (or absent that, will be dynamically allocated by Z/XDC). It is used as the output file that Z/XDC's **PRINT** command writes to. It can be any sequential file, PDS member or spool file. It can have any reasonable DCB attributes. For more information, see **HELP COMMANDS PRINT.**

- **Within the XDCADATA filtering utility,** this ddname is used as the report output file. For more information, see **HELP MAPS ADATA EXCESSADATA XDCADATA.**

Help DDnames PROf

//xxxPROF DD DSN=profilelibrary
//ISPPROF DD DSN=profilelibrary
These are ddnames which, when present, Z/XDC will search for user profiles. They will be searched in the order listed above. For more information, see HELP PROFILES DDNAMES.

Help DDnames Quick

//xxxQUICK DD DUMMY
This is a keyword ddname. Its purpose is to signal the xxxCALL utility that it should not trap execution to Z/XDC before starting the user's program. Instead, the

user's program should just be allowed to start up immediately. For more information, see HELP XDCCALL DDNAMES.

This ddname will be recognized only when a debugging session is started by the xxxCALL utility. For more information, see HELP XDCCALL.

This ddname can be automatically allocated by Z/XDC's Startup Panel in ISPF. Type HELP DEBUGGING ISPF PANEL for more information.

Help DDnames REFr8

//xxxREFR8 DD DUMMY

This is a keyword ddname. Its purpose is to signal the xxxCALL and xxxCALLA utilities that the load modules for the program being debugged must be loaded into key 8 storage even if they are refreshable (REFR) and even if they reside in APF libraries. (xxxCALL[A] accomplishes this by causing TCB_REFRPROT_OVERRIDE to be set on prior to the start of the debugging session.) For more information, see HELP DEBUGGING REFRESHABLE.

This ddname will be recognized only when a debugging session is started by the xxxCALL or xxxCALLA utility. For more information, see HELP XDCCALL.

This ddname can be automatically allocated by Z/XDC's Startup Panel in ISPF. Type HELP DEBUGGING ISPF PANEL for more information.

!!!**WARNING**!!! In spite of the fact that xxxCALLA recognizes and acts on the presence of the **//xxxREFR8** ddname, it generally is **not** a good idea to use it for programs that run authorized. This is because:

- When running authorized, Z/XDC doesn't need modules to be loaded into key 8 storage.
- Without //xxxREFR8, refreshable modules will be loaded into **fetchable** key 0 storage, while with //xxxREFR8, those modules will be loaded into fetch **protected** key 8 storage. This means (among other things) that if your authorized program should switch out of execution key 0 or 8 into any other key, it would **instantly fail** with an s0C4 on the very next instruction fetch.

So while //xxxREFR8 can be immensely helpful for debugging programs running under xxxCALL, it can lead to a variety of unexpected failures for programs running under //xxxCALLA.

Help DDnames RENt8

//xxxRENT8 DD DUMMY

This is a keyword ddname. Its purpose is to signal the xxxCALL and xxxCALLA utilities that the load modules for the program being debugged must be loaded into key 8 storage even if they are reentrant (RENT) and even if they reside in APF libraries. (xxxCALL[A] accomplishes this by causing TCBTCP to be set on prior to the start of the debugging session.) For more information, see HELP DEBUGGING REENTRANT.

This ddname will be recognized only when a debugging session is started by the xxxCALL or xxxCALLA utility. For more information, see HELP XDCCALL.

This ddname can be automatically allocated by Z/XDC's Startup Panel in ISPF. Type HELP DEBUGGING ISPF PANEL for more information.

!!!WARNING!!! In spite of the fact that xxxCALLA recognizes and acts on the presence of the **//xxxRENT8** ddname, it generally is **not** a good idea to use it for programs that run authorized. This is because:

- When running authorized, Z/XDC doesn't need modules to be loaded into key 8 storage.
- Without //xxxRENT8, refreshable modules will be loaded into **fetchable** key 0 storage, while with //xxxRENT8, those modules will be loaded into fetch **protected** key 8 storage. This means (among other things) that if your authorized program should switch out of execution key 0 or 8 into any other key, it would **instantly fail** with an s0C4 on the very next instruction fetch.

So while //xxxRENT8 can be immensely helpful for debugging programs running under xxxCALL, it can lead to a variety of unexpected failures for programs running under //xxxCALLA.

Help DDnames REXx

//xxxREXX DD DSN=rexxlibraries
This ddname points to a library of REXX programs that can be executed as user written Z/XDC commands via Z/XDC's REXX interface. For more information, see HELP REXX.

Help DDnames SDump

//xxxSDUMP DD DUMMY
This is a keyword ddname. When Z/XDC itself suffers from an unexpected ABEND, then the presence of this ddname will cause Z/XDC's internal recovery routine to attempt to produce a dump (instead of relying upon the System's Recovery/Termination Manager [RTM] to do so).

Z/XDC does this by issuing an SDUMPX macro that requests a dump both of all of common storage and of the home address space's private areas. But whether or not a dump is actually produced depends upon your Data Center's local dump management controls.

The presence of a //xxxSDUMP allocation will (at Z/XDC failure time) causes an SDUMP to be requested **regardless** of whether Z/XDC is running authorized or non-authorized.

Note, this ddname is never OPEN'd. If it happens to point to an actual dataset, that dataset will be ignored. For more information, see **HELP SUPPORT SENDINGUSDUMPS.**

Help DDnames XClud

//XDCXCLUD DD DSN=excludefile

This ddname is recognized by the **XDCADATA** program. When present, it contains an **exclusion list of dsects** (and csects) **to be removed from the ADATA.**

//XDCXCLUD is optional. If omitted, then exclusions by name do not occur.

The exclusion list is a sequential, FB-80 dataset or library member. It is built as follows.

Data Records

Each data record must contain one or more **names of dsects** (and csects if desired) to be removed from the ADATA.

The names must be blank separated and located within the first 72 columns of each record.

The last 8 columns (the "sequence number field") are ignored.

Duplicate names are reported but otherwise ignored.

Comments

Comments are ignored.

Comment fields or records start with a comment signal and continue to the end of the record.

In other words, end-of-records are the only end-of-comment signals.

Comments starter signals are any of: * or ; or .* or // or .*

Entirely blank records also are ignored.

Example

```
//XDCXCLUD DD *

.*********************************************
.* Exclude ADATA related dsects               *
.*********************************************
ADATA_AID    ADATA_AOPT   ADATA_COMP
ADATA_DCDS   ADATA_DCX    ADATA_ESD    ADATA_JID
ADATA_MACH   ADATA_MXREF

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Exclude SDWA related dsects                   ;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
SDWA         SDWANRC1     SDWANRC2     SDWANRC3
SDWAPTRS     SDWARC1      SDWARC2      SDWARC3
SDWARC4      SDWARC5

//////////////////////////////////////////////////
// Exclude Misc other dsects                    //
//////////////////////////////////////////////////
AAUTLSAW     AAUTLSES     AAUTLSTG     ABDPL
ACC          ACCDSCT      ACEE         ACEEDSCT
ACEROD       ACERWD       ACHKLIST     ACSPMD
ACTRC     /* From here to column 80 is a comment
CVTFIX       IHADCB
RBPRFX       TCBFIX
/*
```

The Starter List

Z/XDC comes with a "starter list" of nearly 2,300 names of z/OS dsects that you may want to exclude from your ADATA. You may want to do this because these maps are available from a repository of z/OS maps that is installed as part of the Z/XDC product. The repository is named **XDCMAPS.** See **HELP MAPS XDCMAPS** for more information.

The starter list is named **ZEXCLUDE** and is found in **hlq.XDCV31.XDCSAMP.** (See **HELP SUPPORT HLQ** for information about Z/XDC Product Library names.)

If you wish to append your own lists of exclusions, you can do so via JCL concatenation. Example:

```
//XDCXCLUD DD DISP=SHR,DSN=hlq.XDCV31.XDCSAMP
//         DD *
name1 name2 name3     /* your custom exclusion list
/*
//         DD DISP=SHR,DSN=your.standard.exclusion.list
```

Help DDnames SYSAdata

//SYSADATA DD DSN=adatafile
This ddname is recognized by IBM's High Level Assembler (not by Z/XDC). When PARM=ADATA is specified to the Assembler, the Assembler will write ADATA information to this file. Then if the **asm/XDC Feature** is licensed, then Z/XDC's **MAP** command can be used to read this data and build Assembler source image maps with it.

The ADATA file must be sequential, and its DCB attributes must be VB-8188 (or larger). For more information, see IBM's **High Level Assembler Programmer's Guide** (SC26-4941).

Help DDnames SYSCdbg

//SYSCDBG DD DSN=dwarfdatafile
This ddname is recognized by the XL C/C++ and Metal C compiler (not by Z/XDC). When PARM=DEBUG is specified to the compiler, the compiler will write DWARF information to this file. Then if the **c/XDC Feature** is licensed, then Z/XDC's **MAP** command can be used to read this data and build XL C/C++ and Metal C source image maps with it.

The DWARF data file must be sequential, and its DCB attributes must be FB-80. For more information, see IBM's **XL C/C++ User's Guide** (SC09-4767).

Help DDnames SYSMdump

SYSMDUMP DD DSN=sequentialfile
This is a standard z/OS allocation that can be added when the user wants to obtain a storage dump when an unrecovered ABEND occurs.

The dump is written as binary data, so it is not human readable. You will have to use a special utility in order to examine it:

IPCS - This is an IBM provided utility that offers a powerful command set specifically (and only) for examining dumps.

dump/XDC - This is a ColeSoft provided utility that offers its powerful command set for examining **both** dumps and live storage. If you already know how to use Z/XDC, you can use those skills to examine dumps without also having to know IPCS.

Both - While there is some overlap between IPCS and dump/XDC, there also is a lot that is uniquely powerful about each of them. So if you're fortunate enough to possess both **skill sets,** then you will have the best of both worlds.

SYSMDUMPS tend to contain limited information as compared to other IPCS dumps. In particular, they focus on gathering user data in private area storage sections, leaving much of common storage behind. dump/XDC compensates for these omissions by making extractions from host system storage where both appropriate and possible. Consequently, dump/XDC works best against SYSMDUMPs when used on the same system from which the dump came. For more information, see **HELP DUMPS EXPECTEDUSES PERSONALUSE SYSMDUMPS.**

Help DDnames SYS000nn

SYS000nn DD SYSOUT=class
SYS000nn DD DSN=sequentialfile
Unless suppressed by the user, Z/XDC will write a copy of its session log to a sequential output file. By default, this file will be written to spool, but it can

be overridden to be written to disk instead. In any case, Z/XDC always dynamically allocates this file, so the user cannot preallocate it via JCL (nor via TSO's ALLOCATE command).

When allocating this file, Z/XDC does not specify a ddname, so the system will construct a name having the form: SYS000nn. For more information, see HELP FULLSCREEN LOGGING.

Help DDnames TAsklib

//TASKLIB DD DSN=loadlibraries
//anyname DD DSN=loadlibraries
//xxxT0006 DD DSN=loadlibraries

The TASKLIB ddname points to one or more load libraries that contain load modules for the program to be debugged. For more information, see **HELP XDCCALL TASKLIB.**

The xxxCALL utility provides a way in which the name TASKLIB can be changed to any other ddname. Again, for more information, see **HELP XDCCALL TASKLIB.**

The TASKLIB ddname (and its variations or substitutes) will be recognized only when a debugging session is started by the xxxCALL utility. For more information, see **HELP XDCCALL.**

When the tasklib is allocated by Z/XDC's Startup Panel, the ddname that it will use is **//xxxT0006.** Type **HELP DEBUGGING ISPF PANEL** for more information.

xxxCALLA and **xxxCMDA** (for debugging **authorized** programs and TSO commands, respectively) have an added feature in their support for task libraries. They will cause the task library concatenation to be treated as being APF authorized regardless of whether or not the libraries within the concatenation actually are authorized.

This is of great convenience to the developer, but it also creates security concerns. However, the security issues can be controlled by the proper creation of Z/XDC related security rules. See **HELP SECURITY** for complete information.

For example, using system security rules, you can restrict who is and isn't allowed to use the xxxCALLA and xxxCMDA load modules.

You also can restrict who is allowed to load from the libraries containing the xxxCALLA and xxxCMDA load modules.

Using Z/XDC related security rules, you can restrict who is and is not allowed to run authorized debugging sessions. (See **HELP SECURITY AUTH.)**

Help DDnames TLib

//xxxPLIB DD DSN=panellibrary

//ISPPLIB DD DSN=panellibrary

//xxxTLIB DD DSN=tablelibrary
//ISPTUSR DD DSN=tablelibrary
//ISPTLIB DD DSN=tablelibrary
When Z/XDC is using ISPF Display Services to paint its terminal displays, when these ddnames are present, then Z/XDC will search these first (in the order listed) for the template panel definitions (xxxV31A and xxxV31B) that it uses for building its displays.

When the //xxx--- ddnames are absent, Z/XDC will still search the //ISP--- ddnames. For more information, see HELP FULLSCREEN ISPF.

These ddnames can be provided regardless of whether the debugging session is running within TSO or within the batch.

Help DDnames TRAce

//xxxTRACE DD SYSOUT=class
This ddname activates Z/XDC's internal terminal communications trace. This is a low level trace of all of the actions that Z/XDC takes to connect to a user's terminal, send displays to it, and receive commands and other fullscreen actions back.

The trace is a text output file that Z/XDC writes to the sequential file (usually spool) designated by the ddname. It can have any reasonable DCB attributes.

For more information, see HELP FULLSCREEN TERMINALS PROBLEMS.

Help DDnames TRSaf

//xxxTRSAF DD DUMMY
This is a keyword ddname. Its purpose is to activate Z/XDC related security call tracing immediately upon the start of a debugging session.

Normally, Z/XDC's security trace can be turned on or off by the **SET SECURITY TRACE/NOTRACE** command. However, some important security decisions are made during the initialization phase of a debugging session, prior to any possibility of issuing a **SET SECURITY** command. So this ddname provides a way for a debugging session to start up as if a **SET SECURITY TRACE** command. had been issued.

For more information, see HELP COMMANDS SET SECURITY.

Help DDnames TSprt

//xxxTSPRT DD SYSOUT=class

//SYSTSPRT DD SYSOUT=class
These ddnames are used by Z/XDC's REXX interface. (See HELP COMMANDS REXX.) When a REXX program is running within the interface, any output generated either by REXX's TRACE instruction or SAY instructions will be written to one or the other of these ddnames. The rules are as follows:

In TSO:
- SAY/TRACE output will be written to **//SYSTSPRT.**

In the batch:
- If **//xxxTSPRT** is preallocated, then it will receive REXX's SAY/TRACE output.
- Otherwise, if **//SYSTSPRT** is preallocated, then it will receive REXX's SAY/TRACE output.
- Otherwise, the REXX interface will dynamically allocate **//xxxTSPRT DD SYSOUT=*** for receiving REXX's SAY/TRACE output.

Help DDnames Wtor

//xxxWTOR DD DUMMY
This tells Z/XDC that it is allowed to use WTO/WTORs to communicate via System Operator consoles.

If **//xxxWTOR DD DUMMY** and **//xxxCDF DD DUMMY** are **both** present, WTO/WTOR becomes the fallback interface. cdf/XDC is tried first. The WTO/WTOR interface is tried only if cdf/XDC fails for some reason. (Note, one reason would be Operations replying **C** to cdf/XDC's **DBC640Q** query.)

If Z/XDC is directed to use WTO/WTORs, it first issues a WTOR **(DBC829Q)** asking the System Operators for permission to proceed with a debugging session. If Operations replies **NO,** then Z/XDC terminates and percolates the ABEND to the next older ESTAE, ESTAI, or ESTAEX routine.

On the other hand, if Operations replies **YES,** then Z/XDC commences to send its displays to the operator consoles and to accept commands via replies to its WTORs.

For more information, see **HELP USERINTERFACES WTOWTOR.**

Note, when **dump/XDC** is active, **//xxxWTOR DD DUMMY** is ignored.

Help DDnames XDcis

//XDCISxxx DD DUMMY
Now here's a twist, a ddname where the clone name is the **last** three characters not the first. Further, the first three characters **must** be **XDC,** not anything else.

As noted elsewhere, sometimes there are reasons for which you would want to rename Z/XDC to some 3-character name other than **XDC.** (See HELP XDCCLONES for more information about why and how to do this.)

But renaming Z/XDC will create a problem for user programs that build Z/XDC interfaces internally. (See HELP DEBUGGING EXITROUTINES for one discussion about doing this.) If Z/XDC has been renamed to some other 3-character name, then a **LOAD EPLOC==CL8'XDC'** macro will fail to find that correct Z/XDC.

Implementing a scan for a **//XDCISxxx** ddname would be one easy way out of this problem. Just write code that scans the current TIOT for a ddname that starts with **XDCIS.** Then use the ddname's next 3 characters as the name to use when LOAD'ing the XDC load module.

What luck! There just happens to be both a macro and a function subroutine that do exactly that: Scan the TIOT for an XDCISxxx allocation and extract the **xxx** value:

- The macro is named **#XDCIS.** It stores the extracted clone name into an 8-character buffer, padded on the right with five blanks. See HELP XDCCLONES AXDCIS for more information.

- The function subroutine is named **AXDCIS** for Assembler programs and **CXDCIS** for C programs. It returns the 3-character clone name as its returned value (pointed to by R15 for Assembler programs). See HELP XDCCLONES CXDCIS for more information.

Z/XDC itself uses the //XDCISxxx ddname in a couple of places where it needs to find Z/XDC's primary load module (normally named XDC) but where it has no way of detecting whether or not it has been renamed. Those places are:

Z/XDC's JES2 Debugging Support Routines

These routines include a HASPXIT0 exit routine that needs to know what clone of Z/XDC needs to be loaded. When the //XDCISxxx ddname is present, that routine will load xxx, instead of XDC. For more information, see HELP DEBUGGING JES2.

Z/XDC's CICS Debugging Support

This support includes a CICS transaction named XDC. This transaction needs to know what clone of Z/XDC needs to be loaded. When the //XDCISxxx ddname is present, that transaction will load xxx, instead of XDC. For more information, see HELP DEBUGGING CICS.

When Debugging the xxxADATA Filtering Program

If xxxADATA detects a //XDCISxxx allocation, it will activate debugging of itself using the clone of Z/XDC named **xxx.**

Note that absent a //XDCISxxx allocation, the default is to proceed as if **//XDCISXDC** was found (i.e. to use **XDC** as the clone name).

Help DUmps

What is dump/XDC? And what is its Value in an IPCS world?

dump/XDC is a Licensed Feature that allows you to troubleshoot IPCS dumps with the same powerful Z/XDC command set that you use to debug running programs in live storage.

dump/XDC does not replicate IPCS's commands and capabilities, but likewise, IPCS does not replicate Z/XDC's commands and capabilities. Instead, dump/XDC augments IPCS's power with its own considerable abilities. In particular:

- Z/XDC's **MAP** command can be used to display dumped load modules either in source code format or in labeled disassembled format.

- Z/XDC's **DMAP** and **USING** commands can be used to display both system and user control blocks as found in a dump. The cblocks can be displayed either in source code format or disassembled and labeled field by field.

- For **C Programmers,** when the **c/XDC** feature is also licensed, you will be able to see both your C, C++, Metal C and System C source program statements, as well as all your variables, arrays, structures and slices formatted right out of the dump!

- Z/XDC's **LIST** commands can be used to report numerous system structures with its well designed, concise and on point displays.

- Z/XDC's extremely powerful **address expressions** support can be used to chase control block queues, chains and lists like no other tool can, including across whatever dataspaces and address spaces it happens to run through.

- With Z/XDC's **floating address expressions,** you can create floating structures of maps and equates that can be moved from one instance of a control block structure to another simply by rebasing a single, root object. These mapping and labeling structures can be as simple or complex as you need them to be.

- Z/XDC's **autocloning** tools allow you to use a single command to create entire lists of equates or dsects for representing individual elements in lists and chains having typical structures.

In other words, a Z/XDC user can now use the same skill set against a dump that he/she uses when working with running programs in live storage.

For those who are skilled both with Z/XDC and with IPCS, you now have the best of both worlds! dump/XDC runs as an IPCS **verb exit.** It is quite easy to switch back and forth between Z/XDC and IPCS, so both skills can be used while troubleshooting a dump. See **HELP DUMPS STARTINGDUMPXDC SWITCHING.**

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

STARTINGDUMPXDC - Starting up a dump/XDC session.

DUMPTYPES - The types of dumps supported by dump/XDC.

CONTENTS - Storage areas (CSA etc.) and system control blocks (CVT for example) that dump/XDC requires be either available or included in dumps that it processes.

DISAGREEMENTSWITHIPCS - About why dump/XDC may display different registers and PSWs

than IPCS.

EXPECTEDUSES - We expect dump/XDC to be used either for personal use by Assembler and C/C++ developers or for customer support by ISVs.

MAPPINGDATA - This discusses how to enable dump/XDC's access to libraries of mapping data matched to the contents of the dump.

LITERALS - About how dump/XDC uses IPCS literals to preserve information across dump examination sessions.

EXECUTIONTHREAD - About identifying (and changing) the Current Execution Thread in a dump.

COMMANDS - Z/XDC commands that either are only available when using dump/XDC or behave differently when using dump/XDC.

MISCELLANEOUS - Various additional topics.

Current Development Status

At this time (April 2023), some Z/XDC facilities are not available, but we will be adding them as development progresses.

- **System ENQs:** The **LIST ENQ** command is currently disabled in dump/XDC.
- **Access Lists:** The **LIST ACCESSLISTS** command is currently disabled in dump/XDC.

Help DUmps Startingdumpxdc

dump/XDC runs under IBM's IPCS as a **verb exit.** So to use dump/XDC, you first have to start IPCS.

You can run IPCS either within ISPF or from TSO's **READY** prompt or in the **batch.** This topic will step through in great detail how to startup dump/XDC within IPCS running **within ISPF.** Use **HELP DUMPS STARTINGDUMPXDC TSOREADY** for the READY prompt and batch scenarios.

For the following, we will presume:

- That you are using ISPF and IPCS in pretty much the out-of-the-box state that IBM distributes with its **ADCD** builds.
- That the **DXDC** CLIST (and the REXX EXEC that it calls) are available for use by IPCS under ISPF.
- That you've already started **ISPF** within your TSO session.

Let's go.

(1) If it does not already exist, then you need allocate the **IPCSPRNT** ddname. (This will be needed by IPCS's **PRINT** setting. The best place to allocate it to is to a hold class on JESn spool. A **TSO ALLOC FILE(IPCSPRNT) REUSE SYSOUT HOLD** command will do the trick.

No response messages are expected.

(2) Now use **=M.6** to navigate to IPCS's initial panel. In IBM's ADCD build for z/OS, this will be an IPCS version selection panel.

```
--------------- IPCS Primary Panel    ----------------------------
OPTION  ===> IPCS

 IPCS           - IPCS IPL'd operating system
 Z21            - IPCS for z/OS V2 R1    note: execute TSOZ21 at TSO READY
                                               before selecting this
                                               option.
 Z22            - IPCS for z/OS V2 R2    note: execute TSOZ21 at TSO
 [etc.]
```

(3) Issue **IPCS** to start up an IPCS that matches the IPL'd version of z/OS. (Note, there is nothing wrong with starting up older versions of IPCS. dump/XDC works with all versions going back to z/OS R1.6.) The version of IPCS you start should be the the one that matches the version of z/OS running on the system from which the dump came.

First, IPCS issues some linemode messages about which dump directory it allocated.

```
Dump directory name 'DBCOLE.DDIR' will be used
ALLOCATE FILE(IPCSDDIR) REUSE DSNAME('DBCOLE.DDIR') SHR /* @P1C*/
Dump directory 'DBCOLE.DDIR' allocated to FILE(IPCSDDIR)
***
```

(4) When you press **ENTER,** you will finally get to IPCS's actual Primary Options Panel.

```
------------------- z/OS 02.04.00 IPCS PRIMARY OPTION MENU  ------
OPTION  ===> 0

   0  DEFAULTS    - Specify default dump and options
   1  BROWSE      - Browse dump data set
   2  ANALYSIS    - Analyze dump contents
   3  UTILITY     - Perform utility functions
   4  INVENTORY   - Inventory of problem data
   5  SUBMIT      - Submit problem analysis job to batch
   6  COMMAND     - Enter subcommand, CLIST or REXX exec
   T  TUTORIAL    - Learn how to use the IPCS dialog
   X  EXIT        - Terminate using log and list defaults

Enter END command to terminate IPCS dialog
```

(5) At this point, you will want to issue **0** to divert to IPCS's Default Values panel.

```
------------------------ IPCS Default Values --------------------
Command ===>

  You may change any of the defaults listed below. The defaults shown
  before any changes are LOCAL. Change scope to GLOBAL to display global
  defaults.

  Scope   ==> GLOBAL (LOCAL, GLOBAL, or BOTH)

  If you change the Source default, IPCS will display the current default
  Address Space for the new source and will ignore any data entered in the
  Address Space field.

  Source  ==> DSN('DBCOLE.DUMPS.TESTPROG.SYSMDUMP')
  Address Space   ==>
  Message Routing ==> NOPRINT TERMINAL NOPDS
  Message Control ==> CONFIRM VERIFY FLAG(WARNING)
  Display Content ==> NOMACHINE REMARK REQUEST NOSTORAGE SYMBOL ALIGN
```

(6) There are some fields you will want to change on IPCS's Default Values panel:

- First, you may want to change the **Scope** from LOCAL to **BOTH** so that the settings you make here are propagated to all other simultaneous and future instances of IPCS that you wish to start using the same dump directory (DBCOLE.DDIR in this case).

- Second (and most importantly), you will want to fill in the **Source** field with the **dsname** of the dump you want to look at.

(7) When done, use **=6** to navigate to the IPCS Subcommand Entry panel.

```
------------------------ IPCS Subcommand Entry ------------------
Enter a free-form IPCS subcommand or a CLIST or REXX exec invocation below:

===>


---------------------- IPCS Subcommands and Abbreviations -------
ADDDUMP           | DROPDUMP, DROPD   | LISTDUMP, LDMP    | RENUM,
ANALYZE           | DROPMAP,  DROPM   | LISTMAP,  LMAP    | RUNCHA
ARCHECK           | DROPSYM,  DROPS   | LISTSYM,  LSYM    | SCAN
ASCBEXIT, ASCBX   | EPTRACE           | LISTUCB,  LISTU   | SELECT
[etc.]
```

(8) If you need to create **IPCS literals,** now would be a good time to do that.

See **HELP DUMPS LITERALS** for a detailed discussion.

(9) In any case, to start dump/XDC proper, issue **%DXDC.** This invokes a CLIST (which, in turn, invokes a REXX EXEC) for starting the dump/XDC session. (For complete details about the startup process, see **HELP DUMPS STARTINGDUMPXDC DXDCPROC.).**

%DXDC causes IPCS to open and initialize the dump dataset you provided. A series of linemode messages (such as the following) may (or may not) be displayed, describing the dump. For example:

```
IKJ56650I TIME-02:22:51 PM. CPU-00:00:09 SERVICE-9596 SESSION-03:20:11
          SEPTEMBER 17,2020
BLS18122I Initialization in progress for
          DSNAME('DBCOLE.DUMPS.TESTPROG.SYSMDUMP')
BLS18124I TITLE=JOBNAME ABENDS   STEPNAME ABEND64 SYSTEM 0C1
BLS18223I Dump written by z/OS 02.04.00-0 SYSMDUMP - level same as IPCS
          level
BLS18222I z/Architecture mode system

BLS18160D May summary dump data be used by dump access? Enter Y to use, N
          to bypass.
Y
```

(10) The messages may include a query. In this case, **BLS18160D** is asking whether IPCS should include or ignore summary dump information. This matters because summary dump data is captured quite a bit earlier than the rest of the dump, so it may be either more accurate than or simply inconsistent with the rest of the dump. Usually, you should reply **Y** (but not always).

(11) There then may follow some more linemode messages that provide some stats regarding the loading of the dump.

```
BLS18255I Dump Init   Elapsed Time        CPU Time
          Input I/O   00:00:00.032046     00:00:00.013240
          DDIR        00:00:00.015144     00:00:00.014275
BLS18123I 4,812 blocks, 20,017,920 bytes, in
          DSNAME('DBCOLE.DUMPS.TESTPROG.SYSMDUMP')
IKJ56650I TIME-02:36:33 PM.  CPU-00:00:09 SERVICE-9770 SESSION-03:33:53
          SEPTEMBER 17,2020
BLS18224I Dump of z/OS 02.04.00-0 - level same as IPCS level
***
```

(12) When you press **ENTER,** you will finally be taken into Z/XDC proper...

```
------------------------------------ ISPF-XXX-IPCS-VERBEXIT ----
XDC ===>
.  DBC854I z/XDC (with c/XDC, dump/XDC) for z/OS
.  DBC855I COPYRIGHT (C) IZZI SOFTWARE - 2010-2026.  ALL RIGHTS RESERVED
.  DBC856I FOR PRODUCT SUPPORT, PLEASE EMAIL US AT support@colesoft.com
```

```
         .  DBC575I Type ? on the command line to see the Command Dialogs.
         .  DBC993I Type ? at left of any line for a list of its permits...
         .  DBC992I For Built-in Help, type HELP, then press ENTER. (Do ...
         .  DBC994I Type LIST HELP to display the Built-in HELP Index.
         .  DBC996I Type HELP ABOUTHELP for how to use z/XDC's Built-in Help.

         .  DBC353I dump/XDC active. IPCS dump type: 03 (SYSMDUMP) dump/...
         .  DBC354I Title:        JOBNAME ABENDS   STEPNAME ABEND64 SYST...
         .  DBC355I Dsn:          DBCOLE.DUMPS.TESTPROG.SYSMDUMP
         .  DBC356I Original dsn: DBCOLE.DUMPS.TESTPROG.SYSMDUMP
         .  DBC357I Current address space is 002A (ABENDS) (Home)
         .  DBC358I Current execution unit is a TCB (at 7C7D20, RB at 7F...
         .  DBC359I Current execution unit is in ABEND processing, ABEND code: S0C1
                    Reason Code: 00000001 SDWA: 00000000, type: ESTAE(X) (RTM2WA:
                    7F6A0CF0) Handler: 00000000 (Information overridden from ABEND
                    information)
```

(13) At this point, you are in Z/XDC proper with dump/XDC activated for accessing your dump. You can issue most Z/XDC, c/XDC and dump/XDC commands. The information they display will be from the dump. (Try **LIST DXDC.** It might be interesting.)

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

DXDCPROC - Using the **DXDC** REXX procedure to start dump/XDC
PROCESSINGOPTIONS - **Tailoring** dump/XDC's behavior for special situations
SWITCHING - Switching back and forth **between** Z/XDC and IPCS
TSOREADY - Starting up a dump/XDC session from TSO's **READY** prompt
BATCH - Starting up a dump/XDC session when running IPCS in the **batch**

Help DUmps Startingdumpxdc Dxdcproc

The DXDC CLIST

To start your dump/XDC debugging session, you need to navigate to IPCS's **IPCS Subcommand Entry** panel (**=6** from within IPCS) and issue **%DXDC** to run our CLIST and REXX EXEC that start our verb exit (XDCDUMP) that is dump/XDC.

The **DXDC** CLIST (and the REXX EXEC that it calls) must be visible to IPCS. So they have to be copied from their distribution library **(hlq.XDCV31.XDCCLIST)** to a

suitable **//SYSPROC** library on your system. General installation instructions can be found at **HELP MAINTENANCE SUPPORT INSTALLING.**

The **DXDC** CLIST saves the current IPCS PRINT and TERMINAL settings and restores them at the end of the session. This is so that dump/XDC and change them per its own requirements.

If you issue **%DXDC HELP,** the CLIST will display a brief collection of helpful messages and then terminate.

Syntax

The command syntax for starting a dump/XDC session is as follows:

[%]DXDC [DEBUG] [ARGS('parameters')] [CLONE(xxx)]

Notes

Standard TSO CLIST parsing allows all of these operands to be abbreviated down to (in this case) single characters.

Operands

All operands are optional.

%DXDC
This is the startup CLIST's name. The leading % is optional.

DEBUG
During normal processing, dump/XDC is constantly interacting with IPCS, and IPCS frequently tries to issue informational messages that dump/XDC tries to intercept for its own purposes. However, some messages cannot be intercepted and are sent to the terminal as linemode messages. This of course disrupts Z/XDC's fullscreen display.

Most of these linemode messages are unnecessary, so dump/XDC takes steps to suppress them. Unfortunately, occasionally those missed messages might include IPCS error messages that do matter.

So if, for example, your debugging session abruptly ends without explanation, try returning to the session using **%DXDC DEBUG.** This will cause dump/XDC to refrain from suppressing IPCS's linemode messages, and that might be revealing.

ARGS('parameters')
dump/XDC supports several input parameters that you can use to affect its processing

in various ways. Those parameters are documented in **HELP DUMPS STARTINGDUMPXDC DXDCPROC ARGUMENTS.**

CLONE(xxx)

If you are using a **clone** of Z/XDC whose name is other than "XDC", then you can use this parameter to notify dump/XDC what that clone name is. For information about Z/XDC clones, see **HELP XDCCLONES.**

Note that **CLONE(xxx)** can be abbreviated to just **C(xxx).**

If you use **C()** (i.e. provide a null name), DXDC will scan the TIOT looking for a **//XDCISxxx DD DUMMY** and will use whatever clone name is given for "xxx".

Using DXDC in Various Environments

In IPCS under **ISPF,** the **DXDC** CLIST can be invoked from IPCS's **IPCS Subcommand Entry** panel (**=6** from within IPCS).

In IPCS from **TSO READY,** the **DXDC** CLIST can be invoked from the **IPCS** command prompt. Issue TSO's **IPCS** command first, then when you see the prompt, issue **DXDC** to start dump/XDC.

In IPCS **batch,** the **DXDC** CLIST can be invoked from the IPCS command input stream.

The **DXDC** CLIST can only be used within an IPCS environment, as just discussed. If you try using it in other environments (such as from a READY prompt or from elsewhere in ISPF), then an error message will be issued and the CLIST will terminate.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

ARGUMENTS - About the parameters that can be passed to dump/XDC via DXDC's **ARGS('...')** parameter.

Help DUmps Startingdumpxdc Dxdcproc Arguments

Arguments to dump/XDC

You may need to supply arguments to dump/XDC.

These arguments can be supplied using the **ARGS(...)** operand on the DXDC procedure. Note that if more than one value is specified the individual values must be

separated by blanks, and the entire arguments string must be surrounded by single quotes.

Arguments that can be used

UEXIT(name) - This is used to pass a user exit name to Z/XDC. User exits can be used with dump/XDC to perform additional dump analysis.

Note that if this operand is not provided, dump/XDC will attempt by default to load the sample user exit load module: **XDCXITS.**

SDUMP - This indicates that, if XDCDUMP or Z/XDC itself ABENDs, an SDUMP will be taken.

Note that adding a **//XDCSDUMP DD DUMMY** allocation will also cause an SDUMP to be taken when needed.

PROFILE(name) - This names a specific Z/XDC session profile (instead of the default profile) to be loaded going forward. **Name** is the (up to) 5-character name of the profile.

You can use the **LIST PROFILES ALL** command to display the names of all available session profiles.

NORESTORE - For every dump, IPCS maintains a **dump index** that persists across dump/XDC sessions. dump/XDC uses this index for preserving a large amount of mapping, locational and other information across its dump/XDC sessions.

When resuming a dump/XDC session, you can use this parameter if, for some reason, you don't want to restore information learned in your prior dump/XDC sessions.

Important! The information that dump/XDC preserves in a dump's index has nothing to do with and is completely unrelated to the settings that Z/XDC preserves and restores in session profiles.

Help DUmps Startingdumpxdc Processingoptions

dump/XDC processes dumps under the control of a default set of **processing options** that we think is best in most cases. However, if you want to change the options, you may do so by using the keywords described below.

The processing options are conveyed to dump/XDC in an IPCS literal named **DXDCUOPTIONS.** Overrides of the default options need to be set into DXDCUOPTIONS before dump/XDC is started.

- dump/XDC is started from within IPCS by running the **DXDC** REXX procedure. See **HELP DUMPS STARTINGDUMPXDC** for details.

- Use IPCS's **LITERAL** command to set values into literals **before** starting dump/XDC. See **HELP DUMPS LITERALS** for more information.

Note that because the options are stored in an IPCS literal, they are associated with individual dumps. Thus, you may set different processing options for each dump, and there will be no interference between different dumps.

The Syntax:

LITERAL DXDCUOPTIONS C'keyword[,keyword[,...]]'

The Rules:

- **LITERAL** is an IPCS command (not a dump/XDC command). It must be issued before dump/XDC is started.
- The value you assign to the literal must be a list of keywords chosen from the descriptions below.
- The keywords may not be abbreviated from those listed below.
- The keywords must be separated from each other by blanks or commas.
- Leading, trailing and consecutive blanks and commas are ignored.
- IPCS imposes a maximum string length of 256 characters.
- The options string is scanned from left to right.
- Duplicate keywords are ignored.
- When mutually exclusive keywords are given, the last one wins.
- Unrecognized keywords will cause errors to be raised.
- Options whose keywords are not specified remain in effect unchanged.
- The AS... options affect how the **LIST DXDC** command displays the available address spaces. Regardless of how address spaces are displayed, **any** address space on the dump can be used by any Z/XDC command. However, an address space may have missing control blocks, and as a consequence, some Z/XDC commands may fail.

The Options

The following option keywords are recognized. Individual groups of mutually exclusive options are grouped together. Some options have alternate spellings.

IGNORE - This option is ignored. [Alias: IGN]

ASDEFAULT - dump/XDC will select one of the following three processing options based upon the type of dump that's being processed. See **HELP * ASDEFAULT** for an expanded discussion.
[Aliases: ASDF ASDFLT]

ASRTCT - Only address spaces in the dump that are also listed in the dumped system's Recovery/Termination Control Table (RTCT) will be marked as 'selected'. This option is suitable for dumps created by the DUMP

System Operator command.

ASAX1 - Only address spaces where dump/XDC can retrieve the ASXB will be marked as 'selected'. This is suitable for SLIP dumps, SVC dumps and other dump types that are not stand-alone dumps and DUMP command dumps.

ASALL - All address spaces in the dump will be marked as 'selected'. This option is suitable for stand-alone dumps.

ALCHK - Datasets and libraries named in various dump/XDC related IPCS literals will be pre-allocated to ensure that they exist and are catalogued. This is the default.

Such datasets and libraries mainly contain various types of data needed for mapping.

NOALCHK - No dataset allocation pre-checking will be done. They will not be allocated until needed by the **MAP** and **DMAP** commands. If they cannot be allocated at that time, the command will fail.

MDCHK1 - MMEF files (if present) will be pre-opened, and the format of the dataset and the header will be checked. This is the default.

NOMDCHK1 - MMEF files will not be pre-opened.

Notice! When NOMDCHK1 is set, the remaining MDCHKn controls will be ignored. See **HELP * MMEF** for an expanded of Module Mapping Extraction Files.

MDCHK2 - A MMEF's CPUID will be checked against the dump's CPUID and an error raised if they are different. This is the default but will be ignored when NOMDCHK1 is set.

NOMDCHK2 - A MMEF's CPUID will not be checked against the dump's CPUID.

MDCHK3 - A MMEF's OS name will be checked against the dump's OS name and an error raised if they are different. This is the default but will be ignored when NOMDCHK1 is set.

Note, the OS name is obtained from the ECVTPNAM field. For the past decade or two that field has been set to **z/OS.**

NOMDCHK3 - A MMEF's OS name will not be checked against the dump's OS name.

MDCHK4 - A MMEF's OS version will be checked against the dump's OS version and an error raised if they are different. This is the default but will be ignored when NOMDCHK1 is set.

NOMDCHK4 - A MMEF's OS version will not be checked against the dump's OS version.

SUMMSTG - dump/XDC will use IPCS Summary Data when analyzing the dump. This is the default. See **HELP * SUMMARYDATA** for an expanded discussion of a dump's Summary Data.
[Alias: SUMSTG]

NOSUMMSTG - This prevents dump/XDC from using IPCS Summary Data.
[Alias: NOSUMSTG]

AB - This option is no longer valid. See the section on previously valid options below.
[Alias: .AB]

NOAB - This option is no longer valid. See the section on previously valid options below.
[Alias: .NOAB]

ABRB - If the CXU nominated is a task, dump/XDC will locate the most recent ABENDed RB older than the nominated (or defaulted) RB and the CXU will be set to that RB. The number of RBs examined is set by the **DXDCUMAXARBBU** IPCS literal. For more information on this literal, see **HELP DUMPS LITERALS** . This is the default.
[Alias: .ABRB]

NOABRB - The most recent ABENDed RB will not be located.
[Alias: .NOABRB]

ABI - This option is no longer valid. See the section on previously valid options below.
[Alias: .ABI]

NOABI - This option is no longer valid. See the section on previously valid options below.
[Alias: .NOABI]

FRRI - If an FRR (functional Recovery Routine) is active, the PSW and registers from the FRR's SDWA will be used as the PSW and register values for the Current eXecution Unit (CXU). This is the default.
[Alias: .FRRI]

NOFRRI - regardless of whether or not an FRR is active, the current PSW and register values will be used for the CXU.
[Alias: .NOFRRI]

HOVR - Instructs dump/XDC to override the PSW, general registers, access registers and control registers from the dump with corresponding information from the dump header (if said information is relevant to the dump type).

NOHOVR - Prevents dump/XDC from overriding the PSW and registers with information from the dump header. This is the default.

PSWREGS - Instructs dump/XDC to override things (maybe the PSW, maybe CR3 and CR4, maybe the general registers, and maybe the access registers, depending on what was specified) from the PSWREGS=... area if the PSWREGS=... parameter was used on the SDUMP[X] macro. This is the default.

NOPSWREGS - Prevents dump/XDC from overriding things from the PSWREGS=... area even if the PSWREGS=... parameter was used on the SDUMP[X] macro.

PSWREXTS - If the PSWREGS=... area location as indicated by IPCS appears to be incorrect, dump/XDC will try other locations, provided that (1) this option is in effect, and (2) dump/XDC has chosen the CXU (and it has not been overridden by the user). This is the default. Note that this option will be ignored if NOPSWREGS is in effect.

NOPSWREXTS - If the PSWREGS=... area location as indicated by IPCS appears to be incorrect, dump/XDC will not try other locations. Note that this option will be ignored if NOPSWREGS is in effect.

ASISSYS - Allow dump/XDC to decide if the dump system is the same as or different from the current host system. This is the default.

SAMESYS - Always treat the dump system as the same as the current host system.

DIFFSYS - Always treat the dump system as different from the current host system.

DUMPREAD - dump/XDC will directly read the dump dataset to obtain information that is not available via IPCS’s APIs. This is the default.

NODUMPREAD - The dump dataset is not read directly by dump/XDC.

RESTORE - Information that was saved from the last execution of dump/XDC for this dump will be restored. This is the default.

NORESTORE - Information that was saved from the last execution of dump/XDC for this dump will not be restored. Note that if this option is in effect, the **PLAYBACK** and **NOPLAYBACK** options have no effect.

PLAYBACK - Commands that were saved from the last execution of Z/XDC for this dump will be played back. This is the default. Note that if the **NORESTORE** option is in effect, this option has no effect.

NOPLAYBACK - Commands that were saved from the last execution of Z/XDC for this dump will not be played back. Note that if the **NORESTORE** option is in effect, this option has no effect.

PBMSGS - If there are commands that will be played back to restore various things (such as maps and equates) line-mode messages will be issued whilst this is occurring. This is the default.

NOPBMSGS - If there are commands that will be played back to restore various things (such as maps and equates) no line-mode messages will be issued.

NOCPBMSGS - During command playback, Z/XDC **will not** display any messages (including error messages) issued by the played-back commands. This includes not displaying the command itself.

CPBMSGS - During command playback, Z/XDC **will** display all messages issued by the played-back commands. This includes displaying the command itself if the command was issued by the user. This is the default. The messages can be seen by scrolling back.

NOGCMDMSGS - During setup, Z/XDC **will not** display any messages (including error messages) issued by the internally-generated commands. (for example commands generated by the **DXDCUMMEFDSNn** IPCS literals) This includes not displaying the command itself. This is the default.

GCMDMSGS - During setup, Z/XDC **will** display all messages issued by the internally-generated commands (for example commands generated by the **DXDCUMMEFDSNn** IPCS literals). This includes displaying the command itself. The messages can be seen by scrolling back.

Previously valid options

The following options used to be valid options but are no longer valid options. Using them will result in an error. However, to aid in converting them to new options, they are described below and a recommended replacement option is described if relevant.

AB .AB NOAB .NOAB
This option resulted in the extraction of ABEND information from an ABEND SVRB or an FRR. This is now done automatically, thus there is no replacement option.

The ABRB option looks similar, but has a completely different meaning.

ABI .ABI NOABI .NOABI
This option resulted in ABEND information from the RTM2WA (for an ESTAE-type ABEND routine) or SDWA (for an FRR) to be used to supply the PSW and register values.

There is now no need to do this for an ESTAE-type ABEND routine.

The FRRI option can be used to use an active FRR's SDWA information to set the ABEND PSW and register values.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

ASDEFAULT - Expanded discussion of the ASDEFAULT processing option (and friends)

MMEF - Expanded discussion of the processing options pertaining to Module Mapping Extraction Files

SUMMARYDATA - Expanded discussion of a dump's Summary Data

Help DUmps Startingdumpxdc Processingoptions Asdefault

ASDEFAULT - Expanded Discussion

(remember that, regardless of the presence or absence of any AS... options, all address spaces are made available to Z/XDC for processing. The options control the default address space display of the **LIST DXDC** command).

This topic provides additional information concerning the **ASDEFAULT, ASRTCT, ASAX1** and **ASALL** processing options.

- The **ASDEFAULT** option causes dump/XDC to process address spaces according to the dump type. (See **HELP DUMPS DUMPTYPES** for a discussion of dump types.)

- The other **AS<whatever>** options coerce various address space processings regardless of the dump type:

  - **ASALL** marks as 'selected' **all** dumped address spaces. This is the processing option that ASDEFAULT selects for stand-alone dumps.

  - **ASTCT** marks as 'selected' only those address spaces that are listed in the **RTCT.** This is the processing option that ASDEFAULT selects for operator-created dumps (i.e. dumps created by the **DUMP** System Operator used).

  - **ASAX1** is the processing option that ASDEFAULT selects for all dump types other than DUMP-command dumps and stand-alone dumps. It marks as 'selected' only those address spaces for which an **ASXB** can be accessed.

Help DUmps Startingdumpxdc Processingoptions Mmef

MMEF Files - Expanded Discussion

This topic provides additional information concerning the several **[NO]MDCHKn** processing options.

The **[NO]MDCHKn** options control the validity checking of a **Module Mapping Extraction File** (a **MMEF** file).

When a dump is received from a third-party data center (such as from a downstream customer), he/she may send along with the dump an extraction of mapping data for load modules within the dump. This extraction is called the **MMEF** file. It is created by the **XDCMMEF** utility. For more information, see **HELP DUMPS MAPPINGDATA MMEFFILES.**

Help DUmps Startingdumpxdc Processingoptions Summarydata

Summary Data - Expanded Discussion

This topic provides additional information concerning a dump's **Summary Data** and its related processing options.

When a dump is taken, the dumping process obtains a lock and then saves volatile information first. (This is called **Summary Data.)** Then the lock is released, and normal dump processing proceeds.

In due course, the source locations for the Summary Data are reached by the normal dumping process, and the data there is resaved as part of that process. But being volatile, that resaved information may have changed from the point in time it was saved as Summary Data.

Eventually when the dump is examined, there are cases where you need the early data (from Summary Data) and other cases where you're better off with the later data (from the dump). This **[NO]SUMMSTG** option controls which instance of the data is made available to Z/XDC during your dump/XDC session.

Help DUmps Startingdumpxdc Switching

When within dump/XDC, if you wish to issue IPCS commands, there are two ways to do it:

In line:

Issue **Z/XDC's** IPCS command with one or more **IPCS** commands as operands.

Example: Without leaving Z/XDC, you can issue **IPCS 'ANALYZE;LISTDUMP'** to run IPCS's **ANALYZE** and **LISTDUMP** commands. Their outputs will be captured and displayed by Z/XDC, and:

- You will then be able to scroll up and down as you like.
- You will also be able to use Z/XDC's point-and-shoot commands on any hex fields displayed in the IPCS output.

Out of line:

Suspend Z/XDC and return to IPCS by issuing Z/XDC's **END** command (PF keys **F3** or **F4).** Note, in all cases and regardless of the operands used, the **END** command only suspends Z/XDC. It never actually ends the session.

What happens next depends on the environment that IPCS is running in:

- If IPCS is running within **ISPF,** you will now be in the IPCS "Command Entry" panel.
- If IPCS is running from the **TSO READY** prompt, you will now be at IPCS's line-mode prompt.
- If IPCS is running in **the batch,** the next command to be processed will be an IPCS command read from //SYSTSIN.

You can now enter all the native IPCS commands you like. Their outputs will be displayed based on the IPCS environment and your IPCS PRINT and TERMINAL settings.

When you wish to return to Z/XDC, just issue the **%DXDC** command again.

- Your Z/XDC session will be resumed.
- In most cases, all of your maps and equates will be reinstated,
- And your Z/XDC session profile and window settings will be resumed.

Help DUmps Startingdumpxdc Tsoready

dump/XDC is invoked from within IPCS using the **%DXDC** procedure. dump/XDC can be used within all of the following environments:

- IPCS invoked from TSO's **READY** prompt
- IPCS invoked from within **ISPF**
- IPCS running under the TMP (IKJEFT01) in the **batch**

In this topic, I will discuss starting dump/XDC from TSO's READY prompt.

1) From TSO's **READY** prompt, issue **IPCS** to start IBM's IPCS utility.

```
READY
IPCS
```

```
IPCS will start in linemode and display various messages...

    BLS21001I IPCS for z/OS 02.04.00-0
    Dump directory name 'DBCOLE.DDIR' will be used
    ALLOCATE FILE(IPCSDDIR) REUSE DSNAME('DBCOLE.DDIR') SHR /* @P1C*/
    Dump directory 'DBCOLE.DDIR' allocated to FILE(IPCSDDIR)
    BLS21110I CISIZE(22K) is less than 24K. It may degrade IPCS performance
    IPCS
```

Among other things, IPCS will allocate a dump directory (SYS1.DDIR or profileprefix.DDIR). Eventually, it will come to rest at an **IPCS** prompt.

2) At this point, you may wish to issue various IPCS commands, including the following:

- A **SETDEF DSN(dumpdatasetname)** to tell IPCS what dump dataset you want to examine.

```
SETDEF DSN(MYPROJ.SYSMDUMP)
IPCS

SETDEF
/*--------------- Global Default Values for IPCS Subcommands ----------
SETDEF GLOBAL   PRINT NOTERMINAL NOPDS    /* Routing of displays
SETDEF GLOBAL FLAG(WARNING)               /* Optional diagnostic messag
SETDEF GLOBAL NOCONFIRM                   /* Double-checking major acts
SETDEF GLOBAL NOTEST                      /* IPCS application testing
SETDEF GLOBAL   DSNAME('DBCOLE.MYPROJ.SYSMDUMP')
SETDEF GLOBAL LENGTH(4)                   /* Default data length
SETDEF GLOBAL   VERIFY                    /* Optional dumping of data
SETDEF GLOBAL DISPLAY(NOMACHINE)          /* Include storage keys, ....
SETDEF GLOBAL DISPLAY(  REMARK)           /* Include remark text
SETDEF GLOBAL DISPLAY(  REQUEST)          /* Include model LIST subcomm
SETDEF GLOBAL DISPLAY(NOSTORAGE)          /* Include contents of storag
SETDEF GLOBAL DISPLAY(  SYMBOL)           /* Include associated symbol
SETDEF GLOBAL DISPLAY(NOALIGN)            /* Align output to byte
SETDEF GLOBAL ASID(X'002A')               /* Default address space

/*--------------- Local Default Values for IPCS Subcommands ----------
SETDEF LOCAL    PRINT NOTERMINAL NOPDS    /* Routing of displays
SETDEF LOCAL  FLAG(WARNING)               /* Optional diagnostic messag
SETDEF LOCAL  NOCONFIRM                   /* Double-checking major acts
SETDEF LOCAL  NOTEST                      /* IPCS application testing
SETDEF LOCAL    DSNAME('DBCOLE.MYPROJ.SYSMDUMP')
SETDEF LOCAL  LENGTH(4)                   /* Default data length
SETDEF LOCAL    VERIFY                    /* Optional dumping of data
SETDEF LOCAL  DISPLAY(NOMACHINE)          /* Include storage keys, ....
```

```
SETDEF LOCAL  DISPLAY(  REMARK)          /* Include remark text
SETDEF LOCAL  DISPLAY(  REQUEST)         /* Include model LIST subcomm
SETDEF LOCAL  DISPLAY(NOSTORAGE)         /* Include contents of storag
SETDEF LOCAL  DISPLAY(  SYMBOL)          /* Include associated symbol
SETDEF LOCAL  DISPLAY(NOALIGN)           /* Align output to byte
SETDEF LOCAL  ASID(X'002A')              /* Default address space
IPCS
```

3) When you're ready to switch over to dump/XDC, issue:
%DXDC
Z/XDC will start up, and the display will switch into fullscreen mode.

```
------------------------------------ TPUT-XXX-IPCS-VERBEXIT ---
XDC ===>
.  DBC854I z/XDC (with c/XDC, dump/XDC) for z/OS
.  DBC855I COPYRIGHT (C) IZZI SOFTWARE - 2010-2026.  ALL RIGHTS RESERVED
.  DBC856I FOR PRODUCT SUPPORT, PLEASE EMAIL US AT support@colesoft.com

.  DBC575I Type ? on the command line to see the Command Dialogs.
.  DBC993I Type ? at left of any line for a list of its permitted Line Cmds.
.  DBC992I For Built-in Help, type HELP, then press ENTER. (Do not use F1.)
.  DBC994I Type LIST HELP to display the Built-in HELP Index.
.  DBC996I Type HELP ABOUTHELP for how to use z/XDC's Built-in Help.

.  DBC353I dump/XDC active. IPCS dump type: 03 (SYSMDUMP) dump/XDC type:
           SYSMDUMP
.  DBC354I Title:        JOBNAME ABENDS   STEPNAME ABEND64 SYSTEM 0C1
.  DBC355I Dsn:          DBCOLE.DUMPS.TESTPROG.SYSMDUMP
.  DBC356I Original dsn: DBCOLE.DUMPS.TESTPROG.SYSMDUMP
.  DBC357I Current address space is 002A (ABENDS) (Home)
.  DBC358I Current execution unit is a TCB (at 7C7D20, RB at 7FF9F0), and is
           not running
.  DBC359I Current execution unit is in ABEND processing, ABEND code: S0C1
           Reason Code: 00000001 SDWA: 00000000, type: ESTAE(X)  (RTM2WA:
           7F6A0CF0) Handler: 00000000 (Information overridden from ABEND
           information)
```

4) If for some reason IPCS balks at running the verb exit, try issuing IPCS's **DROPDUMP** command, and then try the **%DXDC** command again.

The **DXDC** procedure understands a variety of operands. For the complete syntax along with our recommendations, see **HELP DUMPS STARTINGDUMPXDC DXDCPROC**

Help DUmps Startingdumpxdc Batch

dump/XDC is invoked from within IPCS using the **%DXDC** command. It can be used within all of the following environments:

- IPCS invoked from TSO's **READY** prompt
- IPCS invoked from within **ISPF**
- IPCS running under the TMP (IKJEFT01) in **the batch**

In this topic, I will discuss running dump/XDC in the batch.

IPCS must run as a TSO command. So to run it in the batch, you must start a "TSO batch" environment by running TSO's Terminal Monitor Program (TMP): IKJEFT01.

Batch Processing Characteristics

When running in the batch, both IPCS and dump/XDC run as linemode devices, i.e. they run **non-interactively.**

- They both read canned commands from **//SYSTSIN.**
- They both write all their messages and displays to **//SYSTSPRT.**

In other words, when running in the batch, dump/XDC runs completely script driven.

Note that the xxxWTOR ddname is ignored by dump/XDC.

Note that the xxxCDF ddname is ignored by dump/XDC.

Normally (i.e. when used interactively in TSO), dump/XDC will save and restore both mapping data and session profile data from one dump/XDC session to the next. (See **HELP DUMPS MISCELLANEOUS PROFILES** for more information.)

However, when used in the batch, dump/XDC skips both restoring and saving mapping and profile data. The restoration and saving of such data is strictly reserved for interactive dump/XDC sessions. So batch sessions do not affect interactive sessions, and interactive sessions do not affect batch sessions.

Sample JCL

Here is some sample JCL for running dump/XDC within IPCS.

```
//------- JOB ---
//IPCS   EXEC PGM=IKJEFT01,REGION=0M
//IPCSDDIR DD DISP=SHR,DSN=userid.DDIR
//SYSPROC  DD DISP=SHR,DSN=SYS1.SBLSCLI0   [IPCS CLISTs]
//         DD DISP=SHR,DSN=hlq.XDCV31.XDCCLIST
//SYSTSPRT DD SYSOUT=*
//SYSTSIN  DD *
PROFILE MSGID      /* Don't suppress message numbers  */
IPCS NOPARM        /* Don't use IPCSPRxx from PARMLIB */
SETDEF DSN('adumpdataset') LIST NOCONFIRM /* Sets     */
                   /* dump's dsname. Also suppresses  */
                   /* confirmation WTORs              */

%DXDC              /* Invokes dump/XDC                */

[One or more Z/XDC commands]
```

```
    END                 /* terminate Z/XDC and dump/XDC    */
    END                 /* terminate IPCS                  */
```

Notes About the JCL:

//IPCSDDIR: When present, this allocates the **Dump Directory** that IPCS will use. When omitted, **userid.DDIR** is used. If userid.DDIR does not exist, then it is created.

//SYSPROC: Many of IPCS commands are actually CLISTs or REXX EXECs. SYS1.SBLSCLI0 is a library of such from IBM.

XDC's CLIST library is also concatenated here. The **DXDC** CLIST is loaded from it, along with the REXX EXEC that it calls.

//SYSTSPRT: This is TSO's "standard output" ddname. All TSO, IPCS and Z/XDC messages will be written here.

//SYSTSIN: This is TSO's "standard input" ddname. All TSO, IPCS and Z/XDC commands are read from here.

PROFILE MSGID: This is a TSO command that prevents the suppression of message id numbers for error and warning messages from both TSO and IPCS.

IPCS NOPARM: This is a TSO command that starts IPCS. The **NOPARM** operand prevents IPCS from accessing the IPCSPRxx members of PARMLIB.

SETDEF DSN('adumpdataset') LIST NOCONFIRM: This is an IPCS command that identifies to IPCS the dump dataset to be processed. The **NOCONFIRM** operand prevents IPCS from issuing WTORs asking for confirmations of significant actions. Instead, a default action is taken.

- **%DXDC** is the CLIST that eventually causes dump/XDC to start up.
- The **DXDC** CLIST understands a variety of operands. For the complete syntax along with our recommendations, see **HELP DUMPS STARTINGDUMPXDC DXDCPROC.**

One or more Z/XDC commands: When dump/XDC is invoked in the batch, it causes Z/XDC to process as a linemode device.

- All fullscreen related commands are invalid.
- Z/XDC does **not** run interactively.
- All commands are read from //SYSTSIN.
- All messages and displays are written to //SYSTSPRT.

Flipping In and Out of Z/XDC: By interleaving Z/XDC's **END** command with IPCS's **%DXDC** command, you can flip back and forth between Z/XDC and IPCS. For more information about doing that, see **HELP DUMPS STARTINGDUMPXDC SWITCHING.**

Z/XDC's IPCS Command: You cannot use **Z/XDC's** IPCS command in batch. If you need to issue an IPCS command (while within Z/XDC)...

- Issue Z/XDC's **END** command to pause Z/XDC
- Issue the IPCS commands you want to issue.
- Reissue IPCS's **%DXDC** command to return to Z/XDC.

Help DUmps DUmptypes

dump/XDC can handle any type of dump that IPCS supports. Here, we describe how dump/XDC processes each type.

A dump's header record contains a **dump type** code. In many cases, this is an insufficient indicator of the actual type of dump. Some dumps need further analysis to determine the type to dump/XDC's satisfaction.

Supported Dump Types

The following dump types are supported by dump/XDC:

- **Stand-Alone Dumps:** These are dumps that might be taken after a complete failure of the operating system, when the system is dead and will have to be reIPL'd. It is called stand-alone because the system is so dead that it cannot take a dump on its own. Instead, an entirely separate program has to be IPL'd in order to take the dump.

- **SDUMPs:** These are dumps produced by use of the **SDUMP[X]** Assembler macros. They can be used only by authorized programs.

- **Operator Dumps:** These are SDUMPs produced by the **DUMP** Operator command. Internally, the DUMP command issues the SDUMPX macro to produce the dump. Unlike normal SDUMPs, dump/XDC must perform a deeper analysis to determine (or guess at) what failure caused the developer to request the Operators to create this dump.

- **SLIP Dumps:** These are dumps produced by intentional traps ("SLIP traps") created to capture dumps upon the occurrence of specified events defined by the person who requested the dump. There are several kinds of SLIP dumps. dump/XDC must

perform a deeper analysis to determine which kind the dump is.

- **SYSMDUMP Dumps:** These are dumps produced by the use of the //SYSMDUMP ddname and is an alternative to a SYSUDUMP dump or SYSABEND dump. SYSMDUMPs are the only kind of binary dump that can be created by a problem state program.

  SYSMDUMPs typically will contain mostly private storage data and very little information from common storage. Consequently, there are certain special steps that need to be taken to make it possible for dump/XDC to process SYSMDUMPs. These are discussed further below.

Unsupported Dump Types

The following dump types are **not** supported by dump/XDC:

- **IPCS ACTIVE** Dumps: This is just the means by which IPCS can examine live storage, so in light of the fact that Z/XDC already does that, supporting this "dump" type would be pointless.

- **SYSUDUMPs** and **SYSABEND** Dumps: These are not binary dumps. They are already formatted into human readable text. They are not supported either by IPCS or by dump/XDC.

Processing a Dump Created by the z/OS DUMP Command

If the dump given to dump/XDC was produced by an SDUMP[X] macro, a further analysis is performed to see whether or not it was produced by the Operator's **DUMP** command. If it was, then still more analysis is required.

Normally, the task that issues the SDUMP[X] macro is considered by dump/XDC to be the **current** task. However, for DUMP command created dumps, the "current" task is the dump command itself, not the program for which the dump was requested. So dump/XDC needs to conduct further analysis to discover what the task of interest might be.

But bear in mind that, at best, dump/XDC can only make a guess at this. This is because the DUMP command dumps one or more **entire** address spaces. The problem is, address spaces always contain multiple tasks.

dump/XDC will select the first (or only) aspace dumped as the CAS. If a jobstep TCB can be located via the ASXB in that aspace, it will be set as the CXU.

SLIP Dump Subtypes

If the dump was produced pursuant to a **SLIP** operator command, a further analysis is performed to determine the dump's subtype:

- PER-type SLIP dumps. These are further divided:
  - Instruction fetch (IF)
  - Successful Branch (SBT)
  - Storage alteration (SA)
  - real Storage alteration (SAS)
  - Zero-address detection (ZAD)

- ABEND code ('COMP') SLIP dumps.
- MSGID SLIP dumps.
- ERROR SLIP dumps.

dump/XDC supports all SLIP dump subtypes.

Additional information on SLIP dumps and be found in the help topic **HELP DUMPS DUMPTYPES SLIP**

Determining the Current Execution Thread

When dump/XDC begins processing a new dump, one of the things it tries to do is identify an execution thread in the dump that should be considered as being the **current** thread. Sometimes, that's quite doable, while other times it does not even make sense!

Sometimes (such as for SDUMPs, SYSMDUMPS and SLIPs), dump/XDC can know exactly what thread was current. But sometimes (such as for DUMP command dumps), it can only make guesses. While sometimes (such as for stand-alone dumps), it has to just throw up its hands and not set anything at all. It all depends upon the dump type.

If you want the gory details, check out **HELP DUMPS EXECUTIONTHREAD INITIAL.**

SYSMDUMP Considerations

When processing a SYSMDUMP, several things need to be considered:

- SYSMDUMP datasets can be produced by unauthorized users.
- As a consequence, the information collected by the system and written to a SYSMDUMP dataset is usually restricted (to avoid any sensitive data areas).
- dump/XDC tries to "fill in the blanks" if it detects that the system that you are executing dump/XDC on is the same system that the SYSMDUMP was created on. Where appropriate, when information it needs is missing from the dump, it will look for it in the live system. This is why processing SYSMDUMPs can only be done when dump/XDC is running on the same system on which the dump was created.

For more information, see **HELP DUMPS EXPECTEDUSES PERSONALUSE SYSMDUMPS.**

Help DUmps DUmptypes Slip

One of the dump types that dump/XDC handles is a SLIP dump.

The SLIP operator command is used to set up a SLIP trap.

a SLIP dump is produced as a result of a SLIP trap being tripped, with the action

including:
  SVCD or
  SYNCSVCDUMP
(actions are specified on the SLIP operator command, using the A= or ACTION= operand).

When a SLIP trap is set, the SLIP trap may be:
  A PER SLIP trap. PER stands for Program Event Recording, an interrupt on the z/Architecture hardware. The SLIP command supports the following types of PER interrupts:
    - IF   (Instruction Fetch)
    - SBT   (Successful Branch)
    - SA   (Storage Alteration)
    - SAS   (Real Storage Alteration)
    - ZAD   (Zero Address Detection)
  An ABEND Code ('COMP') SLIP trap.
  A MSGID SLIP trap
  An ERROR SLIP trap.

If the SVC dump was produced as result of the SLIP operator command, the output of the **LIST DXDC** command has a section that displays SLIP information

This section includes a display of the SLIP command.

Other information is displayed, depending on the SLIP command type:
  For PER types, the hardware PER fields are displayed.
  For ABEND (COMP) types, the ABEND information is displayed.
  For MSGID types, message information is displayed.
  For ERROR types, error information is displayed.

In the case of a SLIP trap that tripped as a result of a hardware PER interrupt, please note:
  - The PER interrupt occurs at the **END** of processing of the instruction that resulted in the PER interrupt.
  - As a result the PER interrupt old PSW is pointing at the next instruction to be executed. (If the instruction that caused the PER interrupt is a branch-type instruction the PER interrupt old PSW may be pointing at the branch target address).
  - The PERADDR field has the actual address of the instruction that caused the PER interrupt.

Help DUmps CONtents

dump/XDC uses several system control blocks in its processing. Some of these cblocks are required; if absent, dump/XDC will do one of the following:
  - If the dump was taken on the same system on which dump/XDC is running (i.e. dump/XDC's **host system),** then dump/XDC will take the missing information from the host.
  - Otherwise, dump/XDC will abort.

Other cblocks are not required but are helpful. If they cannot be found, then dump/XDC's capabilities will be limited.

Required System Control Blocks

The following control blocks are required. dump/XDC will be unable to process any dumps for which these are unavailable:
- **ASVT**
- **CPPSA** (PSA for each CPU)
- **CVT** (And extensions)
- **ECVT**
- **GDA**
- **PCCA** (For each CPU in the machine)
- **PCCAVT**
- **PSA** (For the CPU identified in the dump header)
- **RCE**

In the event that one or more of these cblocks is missing from the dump, if the dump was created from dump/XDC's host system (i.e. from the same system on which dump/XDC is currently running), then the instance of the cblock that is present on the host system will be used in its place. (This is expected to happen mainly when processing SYSMDUMPs.)

Optional System Control Blocks

The following cblocks may be used if present. If absent, dump/XDC may be limited in its analysis of the dump:
- **RTCT** (Recovery/Termination Control Table)
- **RTSD** (RTCT SDUMP Extension)
- **TCBs**
- **RBs**
- **SSRBs**
- [etc.]

Providing SDATA Options at Dump Creation Time

In order for a dump to be suitable for processing by dump/XDC, the following **SDATA** options should be provided when creating the dump:
- **ALLNUC** - Entire system nucleus
- **ALLPSA** - The PSAs for all CPUs
- **CSA** - All of CSA
- **LPA** - The PLPA
- **RGN** - The private areas (including LSQA storage) for the address spaces being dumped (Also, see the discussion about address spaces below)
- **SQA** - All of SQA

Other SDATA options may be needed in particular debugging situations, but their presence or absence will not affect dump/XDC processing per se. For example, the person who will be examining the dump may ask you to include **GRSQ** as a dump option, but that will matter to him, not to dump/XDC.

The SDATA options can be provided by:
- The **SDUMP[X]** macro operands,

- The operator **CHNGDUMP** command,
- The operator **DUMP** command,
- Various **PARMLIB** members (for setting system-wide defaults).

For **SLIP** dumps, **DUMP command** dumps and **SDUMPs,** include the following parameter when providing the dump options:
SDATA=(PSA,CSA,LPA,LSQA,RGN,SQA,SUM,SWA,TRT,ALLNUC,GRSQ)

Alternatively, for some dump types, members of **SYS1.PARMLIB** can be used to define default SDATA lists:

- For **DUMP** command dumps, the default SDATA options can be set via the **IEADMCxx** member of SYS1.PARMLIB.

- For **SYSMDUMPs,** the default SDATA options can be set via the **IEADMRxx** member of SYS1.PARMLIB.

  Warning! If you wish to change the areas of storage to be included in SYSMDUMPs, you need to be cognizant of the fact that SYSMDUMPs can be produced by problem state programs, so for security reasons it might not be prudent to include certain areas of common storage in such dumps.

  The default SDATA for a SYSMDUMP specifies NUC, which only dumps part of the nucleus. It is recommended that the ALLNUC option is specified instead, so that all nucleus areas are dumped. This change does not introduce any security exposures.

For **stand-alone dumps,** don’t worry about SDATA. Such dumps automatically dump **everything!**

selecting the address spaces to be dumped

address-space-related SDATA options (for example, RGN), only apply to address spaces that have been selected for dumping. Other address spaces may be included in a dump, but typically only the ASCB and maybe some private storage corresponding to a PSW in some sort of cross-memory mode is dumped.

You can ensure that relevant address spaces are properly dumped by using one of the following techniques:
- If using the **SDUMP[X]** macro, either ensure that the **TYPE=** operand with the **XMEM** or **XMEME** option is specified OR the ASIDs of the relevant address spaces are included in a list of ASIDs pointed to by the **ASIDLIST=** operand of the SDUMP[X] macro
- If using the **DUMP** operator command, ensure that the **XMEMT=YES** operand is in effect.
- If using the **SLIP** operator command, ensure that the **ASIDLST** operand is specified and includes at least the **H** AND **CU** operands. Note that other operands, for example the **SA** operand may be relevant depending on the specific SLIP type.
- If creating a SYSMDUMP, an end-user typically does not have much control over what is included, but a user that can make changes to SYS1.PARMLIB can ensure that the **SDATA** operand includes the **RGN** parameter.
- nothing needs to be done in the case of a stand-alone dump, as all areas of all active address spaces are dumped.

Dumps from Foreign Systems vs the Host System

When a dump is received from a **foreign** system (such as from a downstream customer), it is important that the dump was created under the control of a comprehensive **SDATA** parameter such as discussed above. Otherwise, dump/XDC will not be able to process it. See **HELP DUMPS EXPECTEDUSES ISVUSAGE** for more information.

Note, this means that dump/XDC generally will not be able to process SYSMDUMPs from foreign systems (only from dump/XDC's host system).

On the other hand, when a dump was produced on the same system in which dump/XDC is being used (i.e. dump/XDC's **host** system), then when required information is missing from the dump, dump/XDC will (when appropriate) obtain the corresponding information from the host system. See **HELP DUMPS EXPECTEDUSES PERSONALUSE** for more information.

This is true for all dump types, but it is expected to be especially useful for locally produced **SYSMDUMPs.**

Subtopic Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

FOREIGNVLOCAL - There are special mapping considerations and content requirements that arise for dumps originating from third party systems.

Help DUmps CONtents Foreignvlocal

Where Does the Dump Come From?

dump/XDC needs to know whether the dump it has been given was taken from the **host** system (i.e. the local system on which dump/XDC itself is running) or a **foreign** system. This is so that it can know whether or not certain common storage information that may be missing from the dump can be obtained from the host system. See **HELP DUMPS EXPECTEDUSES** for discussions about where dumps come from.

The host vs. foreign distinction also informs dump/XDC as to whether mapping data for system load modules can be obtained from host system load libraries or overridden with mapping data gathered from the foreign system. See **HELP DUMPS MAPPINGDATA** for discussions about mapping data and where to get it.

Dumps from the Host System

When dump/XDC is given a dump that was created on the local system (dump/XDC's host system), two things are true:

- The **MAP** and **DMAP** commands can use their normal search processes for finding the mapping data it needs.

- When Z/XDC needs information from common storage (the PLPA, the DLPA, the System Nucleus and certain System Control Blocks), if that storage is not in the dump, dump/XDC can find that information on the host system. This assumes, of course, that the system's structure has not significantly changed between the time of the dump and the time of "now" when you are examining the dump.

Notice that this scheme of accessing the host system when necessary is particularly important for **SYSMDUMPs.** That's because such dumps typically do not contain much of common storage, so they likely do not have information that dump/XDC requires in order to process. So without being able to access the host system for information missing from the dump, SYSMDUMPs wouldn't be processable at all! For further discussion, see **HELP DUMPS EXPECTEDUSES PERSONALUSE SYSMDUMPS.**

Dumps from a Foreign System

When a dump was created on a Foreign system, things change a bit...

- Obtaining missing information from host system storage is not an option, so the dump itself will have to contain all the storage data needed by dump/XDC.

  Typically, when we ask our customers to send us a dump, we ask them to specify a rather long **SDATA=** string **(PSA,CSA,LPA,LSQA,RGN,SQA,SUM,SWA,TRT,ALLNUC,GRSQ)** just to be sure the dump contains all the storage that both dump/XDC and the person reading the dump might need. For more details, see **HELP DUMPS DUMPTYPES.**

- Mapping data, on the other hand, is a different story... dump/XDC works best when it has access to mapping data that is matched to the modules, control blocks and data areas contained in the dump. Generally, you will have to ask your downstream customer to provide at least some of that information. For a discussion of identifying what you do and don't need to get, and obtaining what you do need, see **HELP DUMPS MAPPINGDATA.**

  We provide a utility (named **XDCMMEF)** that your customers can run to make the extraction and packaging process of Binder mapping data easier. See **HELP DUMPS MAPPINGDATA MMEFFILES** for more information.

How dump/XDC Decides Whether a Dump is Foreign or Local

There is a series of checks that dump/XDC goes through to make the host vs. foreign determination. Some of these checks can be overridden by various options when you start dump/XDC. (See **HELP DUMPS STARTINGDUMPXDC PROCESSINGOPTIONS.)** But absent overrides, the checks are as follows:

- The CPU id on the dump must be the same as one of the CPU ids for the host system.

- The system name on the dump (normally z/OS) on the dump must be the same as in the host system.

- The system's version and release in the dump must be the same as the host system's.

Help DUmps DIsagreementswithipcs

When dump/XDC has chosen the the **CXU (Current Execution Unit)** or the **CAS (Current Address Space)** (because no user-specified **CXU** or **CAS** settings were present), there may be differences between the values shown by Z/XDC and the values that the IPCS **STATUS** command shows. The **CXU** and **CAS** are discussed in **HELP DUMPS EXECUTIONTHREAD INITIAL.**

These differences can have the following reasons:
- dump/XDC performs a more thorough analysis of a dump than IPCS in an attempt to make an intelligent decision.
- the **[.][NO]ABRB** option.
- dump/XDC's use of other options that cause value overrides.

These reasons are discussed below.

Dump Analysis

Based on the dump type, if no CXU or CAS is specified by the user, dump/XDC may pick a CXU or CAS. For a complete discussion of the dump types supported by dump/XDC, see **HELP DUMPS DUMPTYPES.**

The [.][NO]ABRB option

If the [.]ABRB option is in effect, dump/XDC may choose a CXU that is different from IPCS' choice.

Other dump/XDC options

dump/XDC can use various options that can cause the use of overriding values. Note that an option may be enabled by default.

The following options may be relevant:
- HOVR
- FRRI
- PSWREGS

Subtopics Index

The following additional information is available. Type an **H** at the left to select

directly, or use **HELP *NEXT** (F11) to proceed sequentially. Use **HELP *FORWARD** (F5) to skip.

PSWREGS - Use of the PSWREGS operand on the **SDUMP[X]** Assembler macro.

Help DUmps DIsagreementswithipcs Pswregs

The PSWREGS=... operand on the SDUMP[X] Assembler macro

The SDUMP[X] Assembler macro allows the use of the **PSWREGS=...** operand.

Use of this operand allows the user of the the **SDUMP[X]** Assembler macro to provide an area the contains overriding values for various items (including the PSW and general registers).

when this operand is used, the dump **does not** have the overriding values shown by the IPCS **STATUS** command. The IPCS **STATUS** command will still display the actual values provided to it by the caller of the **SDUMP[X]** Assembler macro.

In fact, within IPCS, displaying or using the overriding values provided by the use of the **PSWREGS=...** operand is quite difficult.

However, dump/XDC recognizes that the **PSWREGS=...** operand has been used and overrides the values appropriately.

dump/XDC will only use the **PSWREGS=...** override if the following circumstances are **all** met:

- This a dump produced by the use of the **SDUMP[X]** Assembler macro, and,
- This is not an operator-requested dump, and,
- dump/XDC has chosen the CXU, and,
- The **PSWREGS=...** operand has been used, and,
- The **PSWREGS** dump/XDC option is in effect (Note that this is the default).

Note that the actual overrides are governed by a set of bits at the front of the **PSWREGS** area provided to the **SDUMP[X]** Assembler macro. For a complete discussion of the format of the **PSWREGS=...** operand, see the appropriate **IBM MVS programming: Authorized Assembler services reference** manual.

Help DUmps EXPecteduses

dump/XDC was written for people who already know how to use Z/XDC and can see a value in being able to apply those skills (either along with or in place of IPCS skills) to investigating storage dumps. So the primary customers for dump/XDC are expected to be those companies who already are customers of Z/XDC.

We further expect that there will be primarily two ways in which dump/XDC will be used:

- **Personal Use:** A developer has a dump (SYSMDUMP usually, or SDUMP or whatever) pertaining to his current work. If he'd like to use Z/XDC to take a look at it, he can do so quite easily, especially if he runs dump/XDC on the same system on which the dump was created.

- **Customer Support Use:** A customer of yours is having trouble with one of your products, so he sends you a dump. You can use dump/XDC to take a look at it.

In order to get the best use from dump/XDC, it needs to have access to mapping data that is **matched** to the dump. In the **Personal Use** scenario, the mapping data will generally be at hand. In the **Customer Support** scenario, your customer may have to send you at least some of the mapping data you will need. For more information, see **HELP DUMPS MAPPINGDATA.**

In the Customer Support scenario, the dumps will often come from a third party system. For a discussion of what dump/XDC needs such dumps to contain, see **HELP DUMPS CONTENTS FOREIGNVLOCAL.**

Regarding Dumps from dump/XDC's Host System

If the dump that you are examining came from the **same** system upon which dump/XDC itself is running (called dump/XDC's **host** system), then there is no problem. You can just let the **MAP** and **DMAP** commands use Z/XDC's normal techniques for finding mapping data. For more information, see:

- **HELP DUMPS EXPECTEDUSES PERSONALUSE**
- **HELP MAPS BINDERMAPS**
- **HELP MAPS SYM**
- **HELP MAPS ADATA**
- **HELP MAPS DWARF**
- **HELP COMMANDS MAP**
- **HELP COMMANDS DMAP**

Regarding Dumps from a Foreign System

On the other hand, if the dump came from a **foreign** system (such as from a **downstream customer),** then you will have to gather at least some mapping data from that system, and you will have to take extra steps to have dump/XDC override Z/XDC's normal map data search process and tell Z/XDC where to find the foreign mapping data. See **HELP DUMPS EXPECTEDUSES ISVUSAGE** for information about foreign systems.

See **HELP DUMPS MAPPINGDATA** about what mapping data needs to be gathered from a foreign system vs what data can be found at hand on the host system.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

PERSONALUSE - Both Assembler programmers **and C programmers** can use dump/XDC to analyze SYSMDUMPs of ABENDs that may occur during the development, testing and debugging of their own personal programs.

ISVUSAGE - We expect that ISVs will find dump/XDC to be highly valuable for troubleshooting SVC dumps, operator command dumps, SLIP dumps and stand-alone dumps that they may receive from their downstream customers.

Help DUmps EXPecteduses Personaluse

The Expected use of dump/XDC by Assembler and C Program Developers

dump/XDC was written for people who already know how to use Z/XDC and can see a value in being able to apply those skills (either along with or in place of IPCS skills) to investigating storage dumps. So the **primary customers** for dump/XDC are expected to be those companies who already are customers of Z/XDC.

While there is some overlap between IPCS and dump/XDC, there also is a lot that is uniquely powerful about each of them. So if you're fortunate enough to possess both skill sets, then you will have the best of both worlds.

But the point is, **now you don't have to know IPCS!** Just by using your Z/XDC knowledge you can now rummage through a dump just as thoroughly as you can through live memory!

C Programmers Can Now Examine Dumps!

During the development, testing and debugging of Assembler and C programs, ABENDs may occur, and dumps may be taken. Of course, if you can easily recreate the failure, you probably already know how use Z/XDC to capture the ABEND and debug it interactively in live storage.

On the other hand, depending upon circumstances, you may find it easier to fire up dump/XDC and take a crack at checking out a dump of the failure.

You already know how to use Z/XDC against live storage. Well, once you get past the setup, you can use the exact same skill set to romp through your dump! After all, storage is storage, right? It may be live, it may be dead, but it's still just storage.

The interesting thing is, historically, dumps have been USELESS to C programmers. However, when using dump/XDC and c/XDC together, that is no longer the case!

- You want to look at your C program in the dump? Go right ahead.
- You want to wander around through your C variables? No problem!
- Do you need to know much about control blocks, registers and PSWs? Not really.

To dig deeper, check out **HELP DUMPS EXPECTEDUSES PERSONALUSE C.**

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

SYSMDUMPS - Creating a dump for analysis via dump/XDC
C - dump/XDC usage for C programmers
ASM - dump/XDC usage for Assembler programmers

Help DUmps EXPecteduses Personaluse Sysmdumps

Why SYSMDUMPs?

There are a couple of interesting things about SYSMDUMPs worth mentioning:

- dump/XDC can be used to analyze any type of dump that IPCS can analyze. However, in general the only kind of dump that a problem state programmer can easily obtain is a **SYSMDUMP.** All other dump types require either supervisor state execution (SDUMPs) or System Operator action (DUMP command dumps, SLIP command dumps and stand-alone dumps).

- While Assembler programmers have benefited for **decades** from being capable of analyzing storage dumps, until now that very valuable diagnostic tool has been **closed to C programmers!** Well, no longer!

  Now, a C programmer with the skills necessary to use c/XDC in their development work also has all the knowledge needed to find and view his/her C programs and C variables contained within a storage dump!

So for personal use, we expect that SYSMDUMPs will be the dump type of choice.

So, What is a SYSMDUMP?

SYSMDUMP is a standard z/OS allocation (ddname) that can be added when the user wants to obtain a storage dump when an unrecovered ABEND occurs. The dump is written as binary data, so you will have to use a special utility (dump/XDC or IPCS) in order to examine it.

To create a SYSMDUMP, the first thing you will need to do is add a **//SYSMDUMP** DD card to your program's runtime JCL. It should create or reference a sequential file...
- With fixed length records
- That are 4160 bytes long,
- And the file needs to be VERY big!

You can use something like the following:

```
    //SYSMDUMP DD DSN=<whatever>,
    // UNIT=DISK<or whatever>,SPACE=(CYL,(100,100),RLSE),
    // DCB=(RECFM=FB,LRECL=4160,BLKSIZE=0),
    // DISP=(NEW,DELETE,CATLG)
```

The second thing is, your program will have to suffer an ABEND that is not recovered.

Interactive Dump Analysis Tools

Ok you've got a dump, and now you want to examine it. The question is, what tools will you use? Here are the main choices...

IPCS - This is an IBM provided utility that offers a powerful command set specifically (and only) for examining dumps.

dump/XDC - This is a ColeSoft provided debugger that offers its powerful command set for examining **both** dumps and live storage. If you already know how to use Z/XDC, you can use those skills to examine dumps without also having to know IPCS.

Both - While there is some overlap between IPCS and dump/XDC, there also is a lot that is uniquely powerful about each of them. So if you're fortunate enough to possess both skill sets, then you will have the best of both worlds.

The Limitations of SYSMDUMPs

One of the problems dump/XDC has to contend with for SYSMDUMPs is that when it comes to common storage, there's very little of it in the dump. This is because SYSMDUMPs focus on capturing user areas of private storage. So system structures (modules and control blocks) that reside in common storage simply are not in the dump.

But dump/XDC needs information from several system control blocks in order to properly understand what's where in the system from which the dump was taken. The needed information includes:

- What load modules are in the Job Pack and Link Pack areas.
- What the boundaries of CSA, SQA and the private areas are.
- What the z/OS release was when the dump was taken. (Sometimes control blocks change shape from one release to the next, and believe me, that matters!)
- And on and on.

Finding the Missing Information

When dump/XDC finds that certain information is missing, it checks to see if the dump was created on the same system in which dump/XDC is running right now (i.e. dump/XDC's **host system).** If so, it makes certain checks to see whether or not the host system still has substantially the same structure as it had when the dump was taken. If so, then dump/XDC will use information from the host when it's not available in the dump.

Issues to Consider for Personal Use Debugging

When using dump/XDC for personal use, you will need to be aware of the following factors:

- The dump you will be examining most likely will be a SYSMDUMP dump. So common storage will largely be absent from the dump.
- You will need to run your dump analysis on the same system on which the dump was taken.
- There must be no significant changes in the shape of common storage between dump time and debug time. In other words:
  - No significant maintenance has been applied to the host system's z/OS between then and now.
  - If an IPL has occurred, it needs to have reloaded all system modules in the same locations they occupied in the prior IPL.
- No changes have been made to the programs and control blocks you are debugging.

If you are running your dump/XDC session on a host system that is **not** the system that the dump came from, then your issues and concerns will be quite different from those described above. For example:
- The dump will have to be a type (any type) other than a SYSMDUMP.
- You may have to obtain at least some mapping data from that system.

See **HELP DUMPS EXPECTEDUSES ISVUSAGE** for the relevant discussions.

Help DUmps EXPecteduses Personaluse C

dump/XDC Usage by C Programmers

Historically, dumps have been useless to C programmers. However, when using dump/XDC and c/XDC together, that is no longer the case!
- You want to look at your C program in the dump? Go right ahead.
- You want to wander around through your C variables? No problem!
- Do you need to know much about control blocks, registers and PSWs? Not really.

So if your program is ABENDing, go ahead and get a dump. Then bring it into dump/XDC, and use your c/XDC skills to see where in your source code the ABEND occurred. You can also rummage around in your C variables to see what's what.

To get a dump, two things need to happen:

- The first thing is, you will need to add a **//SYSMDUMP** DD card to your program's runtime JCL. It should create or reference a sequential file...
  - With fixed length records
  - That are 4160 bytes long,
  - And the file needs to be VERY big!

You can use something like the following:

```
//SYSMDUMP DD DSN=<whatever>,
// UNIT=DISK<or whatever>,SPACE=(CYL,(100,100),RLSE),
// DCB=(RECFM=FB,LRECL=4160,BLKSIZE=0),
// DISP=(NEW,DELETE,CATLG)
```

- The second thing is, your program will have to suffer an ABEND that is not recovered.

Some Assumptions:

- You will be using dump/XDC against a SYSMDUMP, so common storage will largely be absent from the dump.
- You will need to run dump/XDC on the same system on which the dump was taken. Otherwise, the dump will be unusable by dump/XDC.
- There must be no significant changes in the shape of common storage between dump time and dump/XDC time. In other words:
  - No relevant maintenance has been applied to z/OS load modules between then and now.
  - If an IPL has occurred, it has reloaded all system modules into the same locations they occupied in the prior IPL.
- No changes have been made to the programs and control blocks you are debugging.

The nature of SYSMDUMPs

One of the problems dump/XDC has to contend with for SYSMDUMPs is that when it comes to Common Storage, there’s very little of it in the dump. This is because SYSMDUMPs focus is on capturing user areas of private storage. So system structures (modules and control blocks) that reside in Common Storage simply are not in the dump.

But dump/XDC needs information from several system control blocks in order to properly understand what’s where in the system from which the dump was taken. Specifically, dump/XDC needs to know such things as:

- What load modules are in the Job Pack and Link Pack areas,
- What the boundaries of CSA, SQA and the private areas are,
- What the z/OS release was when the dump was taken. (Sometimes control blocks change from one release to the next.)
- And on and on.

Finding the Missing Information

When dump/XDC finds that certain information is missing from the dump, it checks to see if the dump was created on the same system in which dump/XDC is running right now. If so, it will then make certain checks to see whether or not the live system has substantially the same structure as it did when the dump was taken. If so, then dump/XDC will use selected information from the live system when it’s not available in the dump. Consequently:

- You will need to run dump/XDC on the same system on which the dump was taken. Otherwise, the dump will be unusable by dump/XDC.

- There must be no significant changes in the shape of common storage between dump time and dump/XDC time. In other words:
  - No relevant maintenance has been applied to z/OS load modules between then and now.
  - If an IPL has occurred, it has reloaded all system modules into the same locations they occupied in the prior IPL.

- There must be no significant changes made to the programs and control blocks you are debugging.

Help DUmps EXPecteduses Personaluse Asm

dump/XDC Usage by Assembler Programmers

dump/XDC was written for people who already know how to use Z/XDC and can see a value in being able to apply those skills (either along with or in place of IPCS skills) to investigating storage dumps.

dump/XDC does not replicate IPCS's commands and capabilities, but likewise, IPCS does not replicate Z/XDC's commands and capabilities. Instead, dump/XDC augments IPCS's power with its own considerable abilities. In other words, a Z/XDC user can now use the same skill set against a dump that he/she uses when working with running programs in live storage.

For those who are skilled both with Z/XDC and with IPCS, you now have the best of both worlds! dump/XDC runs as an IPCS **verb exit.** It is quite easy to switch back and forth between Z/XDC and IPCS, so both skills can be used while troubleshooting a dump. See **HELP DUMPS STARTINGDUMPXDC SWITCHING** for more information.

The point is, **now you don't have to know IPCS!** Just by using your Z/XDC knowledge you can now rummage through a dump just as thoroughly as you can through live memory!

Getting a Dump

dump/XDC can process any kind of dump that that IPCS can. This includes stand-alone dumps, SLIP dumps, Operator requested dumps (via the DUMP command), SDUMPs and SYSMDUMPS. For further details, see **HELP DUMPS DUMPTYPES.**

The first three of these types require System Operator cooperation. The fourth (SDUMP) can be generated only by authorized programs. Only the fifth (SYSMDUMP) can be generated by problem programs.

(Note, neither dump/XDC nor IPCS can process SYSUDUMPs and SYSABEND dumps.)

Getting an SDUMP

To get an SDUMP, two things need to happen:

- The first is, your program needs to run either APF authorized or in supervisor state or in supervisor key. (Problem-state programs cannot use the SDUMP and SDUMPX macros.)

- The second is, your program's execution must reach an SDUMP or SDUMPX macro.

The dump, by default, is written to a system provided dump dataset. However, you can override that default by providing to the SDUMP[X] macro the DCB of an already OPEN'd output dataset that...
  - Has fixed length records
  - That are 4160 bytes long,
  - And the file needs to be VERY big!

Getting a SYSMDUMP

To get a dump, two things need to happen:

- The first is, you will need to add a **//SYSMDUMP** DD card to your program's runtime JCL. It should create or reference a sequential file...
  - With fixed length records
  - That are 4160 bytes long,
  - And the file needs to be VERY big!

  You can use something like the following:

```
//SYSMDUMP DD DSN=<whatever>,
// UNIT=DISK<or whatever>,SPACE=(CYL,(100,100),RLSE),
// DCB=(RECFM=FB,LRECL=4160,BLKSIZE=0),
// DISP=(NEW,DELETE,CATLG)
```

- The second thing is, your program will have to suffer an ABEND that is not recovered.

The nature of SYSMDUMPs

One of the problems dump/XDC has to contend with for SYSMDUMPs is that when it comes to Common Storage, there's very little of it in the dump. This is because SYSMDUMPs focus is on capturing user areas of private storage. So system structures (modules and control blocks) that reside in Common Storage simply are not in the dump.

But dump/XDC needs information from several system control blocks in order to properly understand what's where in the system from which the dump was taken. Specifically, dump/XDC needs to know such things as:
- What load modules are in the Job Pack and Link Pack areas,
- What the boundaries of CSA, SQA and the private areas are,
- What the z/OS release was when the dump was taken. (Sometimes control blocks change from one release to the next.)
- And on and on.

Finding the Missing Information

When dump/XDC finds that certain information is missing from the dump, it checks to see if the dump was created on the same system in which dump/XDC is running right now. If so, it will then make certain checks to see whether or not the live system has substantially the same structure as it did when the dump was taken. If so, then dump/XDC will use selected information from the live system when it's not available in the dump. Consequently:

- You will need to run dump/XDC on the same system on which the dump was taken. Otherwise, the dump will be unusable by dump/XDC.

- There must be no significant changes in the shape of common storage between dump time and dump/XDC time. In other words:
  - No relevant maintenance has been applied to z/OS load modules between then and now.
  - If an IPL has occurred, it has reloaded all system modules into the same locations they occupied in the prior IPL.

- There must be no significant changes made to the programs and control blocks you are debugging.

Help DUmps EXPecteduses Isvusage

The Expected use of dump/XDC by ISV Customer Support Departments

dump/XDC was written for people who already know how to use Z/XDC and can see a value in being able to apply those skills (either along with or in place of IPCS skills) to investigating storage dumps. So the **primary customers** for dump/XDC are expected to be those companies who already are customers of Z/XDC.

Most of our customers are mainframe software development houses (also known as Independent Software Vendors or **ISVs** for short). But it is not expected they will use dump/XDC much in the development of their own products. (They are already using Z/XDC quite nicely for that, thank you very much).

But when issues arise in an ISV's **own** products running at **its** customer sites (what we refer to as its "downstream customers"), the ISV will need diagnostic information, and often this will include a dump. Debugging these **third party dumps** will likely be an important way developers will use dump/XDC.

Such developers already use Z/XDC's powerful tools in the ongoing development and debugging of their own products. Now dump/XDC will give them the ability to use these same tools when investigating storage dumps.

Obtaining Mapping Data for Third Party Dumps

dump/XDC works best when it has access to load module maps, program maps and cblock maps for the modules and structures reported in a dump. The problem, of course, is

that sometimes the necessary mapping data is located, not at the ISV's site, but at the downstream customer's site. Obtaining that information can be arduous. However, dump/XDC provides a standalone utility **(XDCMMEF)** that the customer can download and run. It scans load libraries, extracts selected mapping data, compresses it and packages it for shipment back to the ISV.

Note that only mapping data is extracted. The load modules themselves are left behind. For more information, see **HELP UTILITIES XDCMMEF.**

More Information

For considerations of what mapping data you may need and how to obtain it, see **HELP DUMPS MAPPINGDATA.**

Help DUmps MAppingdata

dump/XDC Needs Mapping Data that Matches the Dump

When building maps of load modules, csects and control blocks, Z/XDC will use a variety of information from many sources. In order to build accurate maps for objects within a dump, dump/XDC needs to have access to mapping data from the same system from which the dump came.

When an ISV (for example) receives a dump from one of its customers, some of the modules, control blocks and system structures within the dump may be changed from what they are on the ISV's host system. So mapping data, if needed, will have to come from the customer.

The process of locating and loading mapping data will be quite a bit different when troubleshooting a dump from a **foreign system** vs. a dump from the **local system.** This is because when a dump comes from a foreign system, the modules (even system modules) and control blocks (sometimes even system control blocks) within the dump will likely **not** be the same as similar objects on the host system. This is why, when a downstream customer sends you a dump, he/she may also need to send you certain mapping data that matches the dump.

For most mapping data pertaining to your program, it will already be at hand on your own system. But what is **not** always at hand is correct **ESD** data (for building **Binder maps).** This is because when a load module gets relinkedited at the customer's site, the vagaries of the linkedit process can reshuffle the csects within the module.

So for Binder maps, we provide a utility to help with that. It is named **XDCMMEF.** It scans load libraries, extracts ESD data (and SYM data, if any and if requested), compresses it and packages it for shipment back to the ISV. See **HELP UTILITIES XDCMMEF** for more information. Also, read on.

Map Types and the Data Needed to Build Them

When building maps of load modules, csects, control blocks and data areas, Z/XDC will use a variety of information from many sources. Specifically:

- **Binder maps** (i.e. the layout of csects within a load module) are built from the module's **ESD** records.

- **Assembler csect maps** are built from:
  - **SYM data** records contained within load modules,
  - Or **ADATA** records contained within either program objects or (preferably) side files.

- **C program source image maps** are built from a combination of **DWARF data** files and **C program source** files.

When a given category of mapping data is not present, the type of maps dependent upon that data cannot be built.

Ownership Categories

Broadly speaking, modules, control blocks and data areas within a dump fall into three categories:
- Things that are yours,
- Things that are theirs, and
- Things that are IBM's.

How you obtain mapping data may vary amongst these categories.

Things that are Yours

When an ISV needs to examine its own modules, control blocks and data areas contained in a dump, most of necessary mapping data (SYM data, ADATA, DWARF data and C source files) can be found at hand on the ISV's own system.

But what is **not** always at hand is correct **ESD** data (for building **Binder maps).** This is because when a load module gets relinkedited at the customer's site, the linkedit process can reshuffle the csects within the module.

In this case, the customer may have to send load module mapping data (i.e. **ESD** information) to the ISV, but he will **not** have to send **entire load modules** (nor any other mapping data), just the ESD data (and SYM data if any and if requested).

dump/XDC provides a utility specifically for doing this. **XDCMMEF** is a publicly available, stand-alone program that extracts ESD information from load modules (and program objects) and packages it for shipment to the ISV. It can process entire and multiple libraries in a single run!

Both XDCMMEF and its usage doc are publicly available and can be freely downloaded (with no obligation) from our website at **colesoft.com.** For more information, see **HELP SUPPORT ONLINE WEBSITE.**

Things that are Theirs

Sometimes when shooting a problem in a customer provided dump, it can become necessary to examine a customer-written module (an exit routine, for example) more closely. In this case, the ISV may want to request more detailed mapping data (not just the module map data) from the customer. This could include either ADATA or SYM data for program code, for data areas and for control blocks.

The ESD data (and SYM data, if any) can come via the XDCMMEF utility. Any ADATA, DWARF data and C source code needed will have to come from the customer's own files.

Things that are IBM's

Load Modules: IBM load modules typically will vary from system to system due to the application of maintenance or to a difference in z/OS releases or even to just the relinkedit process itself.

While it is of course not expected that anyone outside of IBM will have mapping data for IBM source code, Z/XDC will still need the **load module ESDs** so that it can build accurate load module layouts. The **XDCMMEF** utility can be used by third party customers to gather this mapping data for shipment to the ISV.

The utility and its usage doc are publicly available and can be freely downloaded (with no obligation) from our website at **colesoft.com.** For more information, see **HELP SUPPORT ONLINE WEBSITE.**

Control Blocks: IBM control blocks tend to be quite stable across z/OS releases, but some do change.

ColeSoft maintains at its website **release specific collections** of IBM control block mapping data for all z/OS releases going back to **z/OS R1.6.** See **HELP DUMPS MAPPINGDATA ZOSCBLOCKS** for more information.

Making ADATA, DWARF Data and SYM Data Available to dump/XDC for Building Most Maps

Once you've gathered the dump-matched mapping data you're going to need, you then have to connect it to your dump. In general, for ADATA, SYM data, DWARF data and C source, you can use either **dump/XDC** methods or **Z/XDC** methods:

- **dump/XDC methods** involve setting the names of mapping libraries into **IPCS literals.**
  -
    See **HELP DUMPS LITERALS** for basic information about IPCS literals and the particular literals that dump/XDC creates and uses.
  -
    See **HELP DUMPS MAPPINGDATA LITERALS** for information specific to IPCS literals used for connecting various forms of mapping data to dumps.
-
  An alternative **dump/XDC method** involves using the **DEFINE** command to create overrides to the default search orders used by the **MAP** and **DMAP** commands. See **HELP COMMANDS DEFINE** for more information.
- **Z/XDC methods** involve setting the names of mapping libraries using existing Z/XDC commands:
  -
    For **ADATA** maps, you can use:

- Z/XDC's **SET MAPLIBS** commands, or
- The **ADATALIB=** operand on the **MAP** and **DMAP** commands.

- For **DWARF** data and **C source** maps, you can use:
  - Z/XDC's **LIBRARYLISTS** (LL) commands, or
  - The **DWARFLIB=** and **SOURCELIB=** operands on the **MAP** command.

- For **SYM** data maps, you can use:
  - The **SYMDATALIB=** operand on the **MAP** and **DMAP** commands.

Making ESD Data Available to dump/XDC for Building Binder Maps

For Binder maps, the best way to get the necessary **ESD** data is for your customer to run our **XDCMMEF** program at his site. This utility extracts ESD data from his load libraries and compresses it into **MMEF files.** Then the customer will use FTP to send the file to you along with his dump.

Next, you will need to use dump/XDC specific commands and techniques to connect the MMEF files to your dump.

See **HELP DUMPS MAPPINGDATA BINDERMAPS** for more information.

Persistence - Retaining Mapping Data When a dump/XDC Session is Suspended and Later Resumed

One of the very nice features of dump/XDC is that all Z/XDC information pertaining to one dump:
- Is preserved with that dump from one dump/XDC session to the next,
- Does not affect similar information pertaining to a different dump.

The information that gets automatically saved includes:
- All maps (Binder, csect, dsect, SYM data, ADATA and C source maps)
- All equates
- All session profile settings. This includes:
  - The Watch Windows layouts
  - The function keys
  - Numerous session related settings and controls

This information persistence is achieved through a cooperative combination of two mechanisms:

- **IPCS Literals:** When dealing with a foreign dump, there are two mechanisms for connecting dump-matched mapping libraries to the dump. One is by using **IPCS literals.** (The other is by using the **DEFINE** command.) Once created, IPCS literals are automatically saved by IPCS with the dump. For more information, see **HELP DUMPS MAPPINGDATA LITERALS.**

- **Journaling:** When you use Z/XDC commands to build and assign maps and equates to storage (MAP, DMAP and EQUATE), and when you use commands that affect Z/XDC's knowledge of the libraries from which MAPS are loaded (SET MAPLIBS, LL and DEFINE), Z/XDC **journals** those commands and then saves that journal into IPCS literals so that it may be preserved when the dump/XDC sessions is suspended,

and replayed when the session is later resumed. For more information, see **HELP DUMPS MAPPINGDATA JOURNALING.**

Subtopics Index

The following subtopics delve more deeply into the details of how to create, obtain, organize and preserve the mapping data you will need for matching a dump. Type an **H** at the left to select individual subtopics directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

- **BINDERMAPS** - The customer may need to run **XDCMMEF** to extract and send to you the module mapping data (ESD data) that dump/XDC will need for understanding the layouts of csects within dumped load modules.
- **ADATAMAPS** - When you are shooting a problem in your own code but which is located on a customer's system, the ADATA that you will need will usually be available from your own, in-house libraries.
- **SYMDATAMAPS** - If you prefer to use SYM data, this topic discusses how to create **in-house repositories** of SYM data for your products.
- **DWARFMAPS** - Usually the DWARF data you may need will be available from your own in-house libraries.
- **CSOURCE** - The C source files you may need will be available from your own in-house libraries.
- **LITERALS** - This topic discusses the IPCS literals that dump/XDC uses to hold/remember (from one dump/XDC session to the next) the names of libraries containing dump-matched mapping data.
- **MMEFFILES** - Module Mapping Extraction Files contain ESD data extracted from load modules located on foreign systems. They are used for building Binder maps of foreign system load modules.
- **DEFINE** - The **DEFINE** command creates overrides that cause the MAP and DMAP commands to abandon their normal search processes and instead obtain mapping data from libraries that match the dump being processed.
- **JOURNALING** - dump/XDC's command journaling process allows maps and equates to persist from one dump/XDC session to the next.
- **ZOSCBLOCKS** - We maintain at our website (colesoft.com) collections of **mapping data** for large numbers of z/OS cblocks ranging from z/OS's current release (R3.1) all the way back thru **R1.6!** If you log onto the site, you will be able to download and use the cblock mapping data you will need for whichever level of z/OS your dump came from.

Help DUmps MAppingdata Bindermaps

What are Load Module Maps?

Binder maps and **linkedit maps** and **module maps** are three different names for exactly the same thing: Maps of load modules that identify where, within the module, its control sections (csects) are. Generally, I'll try to use the term **Binder map** throughout the Built-in Help.

When mapping load modules or program objects and the csects within them, the first thing Z/XDC is going to need to figure out is where, within the load module (or program object), its csects are. It does this by building a **Binder map** from the load module's **External Symbol Dictionary** (ESD data, for short).

Nearly all load modules contain ESD data. It's something the Binder automatically creates. So building Binder maps is something the **MAP** command does automatically every time it needs to.

Where Does Dump-Matched ESD Data Come From?

The short answer is, if you want to get ESD data that is matched to your dump, then you will have to get it from your customers.

When your customer has one of your products installed onto his system, supposedly the load modules involved are the same as those on your own system. However, even when your customer has load modules that are "identical" to your own masters, it is often the case that the modules are not truly identical. They may have identical content, but they may have **differing structures.** This is because when a load module is reprocessed by the Binder, the csects within the module may get **shuffled around.** And chances are that if your product was installed at your customer's site via **SMP,** your load modules may have been relinkedited.

When you need to examine a load module, you will usually need the ESD data for that module in order to know what csect you are looking at. When you need the ESD data, it usually doesn't matter whether the load module is yours, theirs or IBM's. If you need to see inside it, you may have to ask the customer to send you its ESD data.

But sending ESD data is not as hard as you might think. You do not need to ask your customers for entire load modules, just the ESD data... Here's how.

About XDCMMEF

XDCMMEF is a utility we publicly provide for free (and without obligation) from our website at **colesoft.com.** For more information, see **HELP SUPPORT ONLINE WEBSITE.**

XDCMMEF is a high-speed, stand-alone batch program that your customer can download and run against his/her load libraries to extract just the mapping data that may be present in the load module or program object. This can include:

- The ESD data
- Optionally any SYM data that may be present
- Optionally any ADATA that may be present

All executable content gets left behind.

XDCMMEF compresses and packages the extracted mapping data into a portable,

sequential, FB=80 file that your customer can then FTP to you along with the dump.

XDCMMEF processes linklist libraries, LPALIB libraries and customer selected load libraries (PDSs and PDSEs alike) all in a single run.

XDCMMEF is available to anyone, not just dump/XDC customers. That means it can be freely downloaded by **your** customers (not just ours).

For download and usage information, see **HELP UTILITIES XDCMMEF.**

When MMEF Files Are Connected To a Dump

When MMEF files are connected to a dump, all attempts by the **MAP** and **DMAP** commands to read mapping data from load libraries are **redirected** to the MMEF files.

- This goes for all attempts to read **ESD** data, **SYM** data and **ADATA.**
- This also goes for all attempts to read from both load libraries **(PDSs)** and program object libraries **(PDSEs).**

Connecting MMEF Files to a Dump - Z/XDC Method

Normally, the **MAP** and **DMAP** commands would look in host resident load libraries when searching load modules and program objects for mapping data (ESD data, SYM data or ADATA). To override and cause the commands to look into MMEF files instead, you need to use the **DEFINE MMEFDSN** command to build a list of MMEF file dsnames. When this list is present, the **MAP** and **DMAP** commands will search the listed MMEF files instead of using its normal methods of opening and reading load libraries. For more information, see **HELP COMMANDS DEFINE MMEFDSN.**

Connecting MMEF Files to a Dump - dump/XDC Method

Under the covers, dump/XDC creates, saves and uses a series of **IPCS literals** whose names start with **DXDC.** The literals whose names start with **DXDCU** are user-settable. You can set values into them via IPCS's **LITERAL** command. See **HELP DUMPS LITERALS** for general information about dump/XDC's **user settable** IPCS literals.

The series of literals that pertain to MMEF files are named **DXDCUMMEFDSNn** (where **n** is a sequence number ranging from **1** to **20).** This series allows you to name up to 20 MMEF files to be available whenever a dump/XDC session is started or resumed against the current dump. Examples:

```
LITERAL DXDCUMMEFDSN1 C'DBCOLE.FROMCORP.MISCLIBS.MMEF'
LITERAL DXDCUMMEFDSN2 C'DBCOLE.FROMCORP.LPALIBS.MMEF'
LITERAL DXDCUMMEFDSN3 C'DBCOLE.FROMCORP.LINKLIST.MMEF'
```

When **DXDCUMMEFDSNn** literals exist, dump/XDC issues the following Z/XDC commands (internally) at the start of the session:

- **DEFINE MMEFDSN firstdsname**
- **DEFINE MMEFDSN seconddsname**
- etc.

Help DUmps MAppingdata Adatamaps

What is ADATA?

ADATA is a form of mapping data created by the Assembler. It contains source code images that Z/XDC can use for formatting storage displays of programs and data areas. For more information, see **HELP MAPS ADATA.**

Where Does Dump-Matched ADATA Come From?

The short answer is, if you already use ADATA in your development process, then you will already have at hand the ADATA that matches your dump.

Remarks Regarding ADATA

The Assembler can generate ADATA and write it to either (or both) of two output files:

- **//SYSLIN:** When **PARM=GOFF(ADATA)** is specified, ADATA is generated and written embedded into a GOFF format object code output file for later inclusion into a final program object. (This is **not** recommended.)

- **//SYSADATA:** When **PARM=ADATA** is specified, ADATA is generated and written to a RECFM=VB side file, completely separate from the program object.

Which is Better, ADATA in Side Files or in Program Objects?

When ADATA is present in a GOFF input file, the Binder will automatically include it into the program object as non-LOAD'able class data. However, writing ADATA into program objects is not recommended because:

- The Binder supports inserting ADATA into program objects (as non-loadable class data) but not into load modules.

- ADATA vastly increases the on-DASD size of the program object. (Its LOAD'd-into-storage size is not affected.)

- ADATA can vastly increase the Binder's processing time when it builds the program object.

- ADATA contains source code images, and you probably don't want to send source code information to your customers.

Overall, it is better to manage ADATA in side files, not program objects. For more information, see **HELP MAPS ADATA CREATING.**

Reducing ADATA Quantities and File Sizes

When multiple assemblies are eventually bound together into a handful of load modules (or program objects), the very nature of the multi-assembly process gives rise to vast amounts of duplicate and unnecessary ADATA. To help deal with this, the Z/XDC product includes a utility that, when used properly, can reduce ADATA quantities by up to **90%** or better! The utility is named **XDCADATA.** It is intended to run in the batch between an assembly step and a Binder step.

XDCADATA receives an Assembler created ADATA or GOFF file as input, removes excess ADATA records, and writes a much reduced output for subsequent processing or use. For more information, see **HELP MAPS ADATA EXCESSADATA.**

Where Should You Put ADATA?

So once you've created (and filtered) an ADATA file, where should you put it?

- In the case of //SYSLIN GOFF files, the ADATA will eventually be stored as non-LOAD'able class data in a program object.

- In the case of //SYSADATA side files, the output from the XDCADATA should be saved into a RECFM=VB PDS.

But What Should the Member Name Be?

When saving ADATA into a library, what should you use as a member name? Well, you'd think there'd be a straightforward answer, but there's not...

You see, when the Assembler creates an ADATA file, it stuffs into it the source code information for **all** csects and **all** dsects encountered in the program being assembled, and this creates a bit of a problem: How should Z/XDC find ADATA for a csect or dsect whose name does not match any membername?

Well, there's a big, long, involved discussion about what to do about that, but the short answer is this:

- 1st Choice: Use the name of the primary csect within the assembly. Z/XDC will search for and through this member first. This will work when mapping the primary csect. It will not work when trying to map any other csect or dsect in the assembly.

- 2nd Choice: Use the name of the load module into which the csect will be included. Z/XDC will search for and through this member second. This will work when trying to map any csect or dsect that occurs within any assembly comprising the load module.

- 3rd Choice: Pick an arbitrary name. A MAP or DMAP command operand **(ADATALIB=)** can be used to cause Z/XDC to search through that member instead of either of the above.

  You can also use this method when your ADATA library is not listed in the active MAPLIBS list.

Historical ADATA

Like most ISV's we provide technical support, not only for current product releases, but also for a number of older releases. So it is often the case that we receive dumps from customers that contain our products at a variety of maintenance and release levels. If we are to have ADATA that's matched to a back-leveled dump, then we need to maintain back-leveled ADATA libraries that match the dumps we are likely to receive.

Sometimes, this is worth the trouble, sometimes it is not. Your mileage may vary.

Connecting ADATA Libraries to a Dump - Z/XDC Method

ADATA libraries are used by the **MAP** and **DMAP** commands for building source image maps of Assembler programs and control blocks. There are two methods by which ADATA libraries are made known to Z/XDC:

- The **SET MAPLIBS** command can build and manage multiple named lists of ADATA libraries. See **HELP MAPS ADATA MAPLIBS** for complete details.

- The **ADATALIB=** operand of the **MAP** and **DMAP** commands can be used to specify an ADATA library not already in the currently active MAPLIBS list. (It will be automatically added to the current list, though.)

Connecting ADATA Libraries to a Dump - dump/XDC Method

Under the covers, dump/XDC creates, saves and uses a series of **IPCS literals** whose names start with **DXDC.** The literals whose names start with **DXDCU** are user-settable. You can set values into them via IPCS's **LITERAL** command. See **HELP DUMPS LITERALS** for general information about dump/XDC's **user settable** IPCS literals.

The series of literals that pertain to ADATA libraries are named **DXDCUMAPLIBn** (where **n** is a sequence number ranging from **1** to **20).** This series allows you to name up to 20 ADATA libraries to be available whenever a dump/XDC session is started or resumed against the current dump. Examples:

```
LITERAL DXDCUMAPLIB1 C'DBCOLE.FROMCORP.DSCOPY.ADATA'
LITERAL DXDCUMAPLIB2 C'DBCOLE.FROMCORP.TAPEMAP.ADATA'
```

When **DXDCUMAPLIBn** literals exist, dump/XDC issues the following Z/XDC commands (internally) at the start of the session:

- **SET MAPLIBS RESET** [clears the active MAPLIBS list]
- **SET MAPLIBS firstdsname**
- **SET MAPLIBS seconddsname**
- etc.

Help DUmps MAppingdata Symdatamaps

What is SYM Data?

SYM data is a form of mapping data created by the Assembler. It contains statement labels and field names that Z/XDC can use during disassembly to label machine instructions and data fields in its formatted displays. For more information, see **HELP MAPS SYM.**

SYM data's main advantage as compared to ADATA is that it is vastly smaller.

Where Does Dump-Matched SYM Data Come From?

The short answer is, if you already use SYM data in your development process, then you will already have at hand the SYM data that matches your dump.

Remarks Regarding SYM Data

When requested (via its **PARM=TEST** option), the Assembler generates and writes SYM data into object output files (**//SYSLIN** files). The object file can be either in the classic format or the newer GOFF format, it doesn't matter.

When present in an object file, the Binder will either include it or ignore it according to whether or not its **PARM=TEST** is specified.

When included, the Binder will add the SYM data as a non-LOAD'able class (if a program object) or as a special record type (for load modules). In either case, the on-DASD size of the load module (or program object) will be substantially increased (although its LOAD'd-into-storage size will not be affected).

Notice! **SMP** does not support setting PARM=TEST on linkedits. This means that it is not possible to pass SYM data into load modules created by an SMP install process.

Removing Unnecessary SYM Data

When multiple assemblies are eventually bound together into a handful of load modules (or program objects), the very nature of the multi-assembly process gives rise to large amounts of duplicate and unnecessary SYM data. To help deal with this, the Z/XDC product includes a utility that, when used properly, can reduce SYM data quantities by up to **90%** or better! The utility is named **XDCSYMED.** It is intended to run in the batch between an assembly step and a Binder step.

XDCSYMED receives Assembler created object files (classic or GOFF) as input, removes excess SYM data entries, and writes the much-reduced output for subsequent processing by the Binder. For more information, see **HELP MAPS SYM EXCESSSYMDATA.**

Where Should You Put SYM Data?

SYM data can be conveyed only by object files, and it can be retained only in load modules (or program objects). Unlike ADATA, there is no such thing as side file support.

So SYM data can reside only in load modules (and program objects). However, since SYM data contains symbol data from your source code, you probably don't want to be shipping that information to your customers. So if you want to use SYM data at all, then you will have to manage two instances of your load modules:

- **During Development:** You will want to:
  - Assemble your code with **PARM=TEST,**
  - Run the object output through an **XDCSYMED** step to remove unnecessary SYM data entries,
  - Build your load modules also with **PARM=TEST.**

- **During Shipment Packaging:** You will want to:
  - Save your final development load modules into a historical archive for use when you receive dumps from customers.
  - If you ship object files, you will want to reassemble them **without** PARM=TEST.
  - If you ship load modules or program objects, you will want to relink them **without** PARM=TEST.

Obtaining SYM Data from Your Customer

If for some reason your customer has SYM data that he needs to send you, then he will have to run the **XDCMMEF** utility to extract it from his load modules, and ship it to you in an **MMEF** file. See **HELP DUMPS MAPPINGDATA MMEFFILES** for more information about MMEF files.

Then when you receive the file, you will have to connect it to your dump/XDC session using the same considerations you would use for accessing dump-matched Binder maps...

Use Z/XDC's **DEFINE MMEFDSN** command to build a list of MMEF file dsnames. When this list is present, the **MAP** and **DMAP** commands will search the listed MMEF files instead of using their normal methods of opening and reading load libraries. For more information, see **HELP COMMANDS DEFINE MMEFDSN.**

Using SYM Data from Your Own Local System

If:
- You have a local library of load modules containing SYM data that matches modules in your customer's dump,
- But you have an MMEFDSN token defined that causes all load module I/O to be redirected to an MMEF file,
- But you want to use the local SYM data library nonetheless...

You can make that happen by using the **MAP** command with two operands added:

- Use the **SYMDATALIB=** operand to cause the **MAP** command to access the load library named by the operand.

- And use the **USELDS** operand to prevent the library I/O from being intercepted and redirected towards the MMEF file.

See **HELP COMMANDS MAP SYNTAX** for more information.

Help DUmps MAppingdata DWarfmaps

What is DWARF Data?

DWARF data is a form of mapping control data created by a C compiler. It contains comprehensive information about the variables used by your C program. It also contains the original dataset names of all of the source files used in the compilation of your program. (It does not contain the source images themselves.)

Where Does Dump-Matched DWARF Data Come From?

The short answer is, if you already use DWARF data in your development process, then you will already have at hand the DWARF data that matches your dump.

Remarks Regarding DWARF Data

DWARF data is created in various ways by the C, C++, Metal C and Systems/C compilers. It can be created...

- In classic z/OS libraries and datasets
- In a USS file system
- On a Windows hard drive

Pretty much anywhere. For the particulars, see **HELP MAPS DWARF.**

The original name of the DWARF data file for your program will already be hard-coded into your Program Object's meta data. So in principle, if you do not move or rename your DWARF file, then the **MAP** command will simply be able to automagically find it when needed.

However, there are circumstances where the recorded DWARF file name is not complete or is otherwise not directly usable. So when this is the case (or when you have moved or renamed the DWARF file), you will need to use Z/XDC's **LIBRARYLISTS** command to redirect the original file name to its currently usable name. For more information, see **HELP DEBUGGING C LIBRARYLISTS.**

Connecting DWARF Data Libraries to a Dump - Z/XDC Method

DWARF data libraries (along with C source files) are used by the MAP command for building source image maps of C programs and for obtaining knowledge about the C variables used by those programs. There are two methods by which DWARF data libraries are made known to Z/XDC:

- The **LIBRARYLISTS** command can build and manage multiple, named lists of DWARF data files and libraries. See **HELP COMMANDS LIBRARYLISTS** for details.

- The **DWARFLIB=** operand of the **MAP** command can be used to specifically identify a DWARF data library not already known to the LIBRARYLISTS structure. (It will be automatically added to the lists, though.)

Connecting DWARF Data Libraries to a Dump - dump/XDC Method

Under the covers, dump/XDC creates, saves and uses a series of **IPCS literals** whose names start with **DXDC.** The literals whose names start with **DXDCU** are user-settable. You can set values into them via IPCS's **LITERAL** command. See **HELP DUMPS LITERALS** for general information about dump/XDC's **user settable** IPCS literals.

The series of literals that pertain to DWARF data libraries are named **DXDCUDWLIBn** (where **n** is a sequence number ranging from **1** to **20).** This series allows you to name up to 20 DWARF data libraries to be available whenever a dump/XDC session is started or resumed against the current dump. Examples:

```
LITERAL DXDCUDWLIB1 C'DBCOLE.FROMCORP.ACENT2.DWARFLIB'
LITERAL DXDCUDWLIB2 C'DBCOLE.FROMCORP.WHATEV.DWARFLIB'
```

When **DXDCUDWLIBn** literals exist, dump/XDC issues the following Z/XDC commands at the start of the session:

- **LL RED DWARF PATTERN=(<valuefromliteral>)**

Help DUmps MAppingdata Csource

Where Does Dump-Matched C Source Come From?

The short answer is, you will already have at hand the C source that matches your C programs in your dump.

Remarks Regarding C Source

The original name of the C source files for your program will already be hard-coded into your Program's DWARF data. So in principle, if you do not move or rename your C source files, then the **MAP** command will simply be able to automagically find them when needed.

However, there are circumstances where the recorded C source file names are not complete or are otherwise not directly usable. So when this is the case (or when you have moved or renamed any of the files), you will need to use Z/XDC's **LIBRARYLISTS** command to redirect the original file names to their currently usable names. For more information, see **HELP DEBUGGING C LIBRARYLISTS.**

Connecting C Source Libraries to a Dump - Z/XDC Method

C source libraries (along with DWARF data) are used by the MAP command for building source image maps of C programs. There are two methods by which C source libraries are made known to Z/XDC:

- The **LIBRARYLISTS** command can build and manage multiple, named lists of C source libraries. See **HELP COMMANDS LIBRARYLISTS** for details.

- The **SOURCELIB=** operand of the **MAP** command can be used to specifically identify a C source library not already known to the LIBRARYLISTS structure. (It will be automatically added to the lists, though.)

Connecting C Source Libraries to a Dump - dump/XDC Method

Under the covers, dump/XDC creates, saves and uses a series of **IPCS literals** whose names start with **DXDC.** The literals whose names start with **DXDCU** are user-settable. You can set values into them via IPCS's **LITERAL** command. See **HELP DUMPS LITERALS** for general information about dump/XDC's **user settable** IPCS literals.

The series of literals that pertain to C source libraries are named **DXDCUCSLIBn** (where **n** is a sequence number ranging from **1** to **20).** This series allows you to name up to 20 C source libraries to be available whenever a dump/XDC session is started or resumed against the current dump. Examples:

```
LITERAL DXDCUCSLIB1 C'DBCOLE.FROMCORP.ACENT2.CEELIB'
LITERAL DXDCUCSLIB2 C'DBCOLE.FROMCORP.WHATEV.CSOURLIB'
```

When **DXDCUCSLIBn** literals exist, dump/XDC issues the following Z/XDC commands (internally) at the start of the session:

- **LL RED CSOURCE PATTERN=(<valuefromliteral>)**

Help DUmps MAppingdata Literals

Matching Mapping Data to Your Dump

When loading maps for modules, control blocks and data areas contained in a dump, you need to use mapping data that is matched to the dump. When a dump is received from a foreign system (such as from a downstream customer), at least some of the mapping data will have to come from that system (not from the host system upon which dump/XDC is running).

Z/XDC's normal methods for finding mapping data expects that the needed data resides in local system and user libraries. But this expectation breaks down for dumps from foreign systems. To address this problem, dump/XDC provides support for using **IPCS literals** for defining overrides to Z/XDC's normal search process.

What are IPCS Literals?

IPCS literals are defined in IPCS (outside of Z/XDC). Once created, IPCS permanently saves its literals with the dump, so they are automatically preserved from one dump/XDC session to the next.

IPCS supports creating literals of various data types. The literals that dump/XDC creates and uses are **string** literals. Each string literal is capable of holding up to 256 bytes worth of arbitrary information.

dump/XDC uses IPCS string literals heavily to create and manage control information both within a dump/XDC session and from one session to the next.

Certain dump supporting Z/XDC commands both set and use IPCS literals under the covers. But you can also use IPCS's **LITERAL** command directly to create literals yourself.

The syntax of IPCS's LITERAL command (in summary) is **LITERAL <name> C'<value>'** For a more detailed discussion, see **HELP DUMPS LITERALS.**

Using IPCS Literals to Define Overriding Mapping Libraries

dump/XDC provides two ways to define libraries that contain mapping data matched to a foreign dump:

- One is by using Z/XDC's **DEFINE** command.
- The other is by USING IPCS's **LITERAL** command to manually create IPCS literals containing the names of the overriding libraries.

What follows is a discussion of manually created literals via IPCS **LITERAL <name> C'<value>'** commands.

When defining overriding libraries, you must create various IPCS literals whose **<names>** must be of the form, **DXDCU<libtype><n>** where:
<libtype> - Indicates the type of library involved (see below).
<n> - Permits multiple libraries of a given type to be defined.

Example: **DXDCUDWLIB3**
- This is a dump/XDC supported literal (DXDC).
- It is a user settable literal (U).
- It defines a DWARF data library override (DWLIB).
- It is the third such library (3) in a series of related definitions.

The literal's **<value>** must be the fully qualified dsname of the library or file containing foreign mapping data.

Identifying Mapping Data Override Libraries

For all of the following IPCS literals, equivalent Z/XDC commands exist that perform functions similar to that which the literals do.

As noted above, IPCS literals pertaining to mapping data libraries and files have names of the form **DXDCU<libtype><n>.** The following **<libtypes>** are recognized by

dump/XDC:

MMEFDSN - This names a Module Maps Extraction File (created by the **XDCMMEF** utility) that contains extracted load module **ESD data** for building module maps. Up to 20 of these files may be defined. Examples:
LITERAL DXDCUMMEFDSN1 C'DBCOLE.MMEF.FROM.GLOCORP'
LITERAL DXDCUMMEFDSN2 C'DBCOLE.MMEF.FROM.ANOTHER.CORP'

For information about Module Maps Extraction Files, See **HELP DUMPS MAPPINGDATA BINDERMAPS.**

To communicate these overrides to Z/XDC, these literals cause dump/XDC to generate and send to Z/XDC commands of the form: **DEFINE MMEFDSN value POS=BACK ID=DXDCn**

MAPLIB - This names an overriding library containing **ADATA.** The library will be used by the **MAP** and **DMAP** commands for building csect and dsect maps. Up to 20 of these libraries may be defined. Example: **LITERAL DXDCUMAPLIB1 C'DBCOLE.MYADATA.LIBRARY'**

To communicate these overrides to Z/XDC, these literals cause dump/XDC to generate and send to Z/XDC a **SET MAPLIBS RESET** command followed by one or more commands of the form: **SET MAPLIBS dsname**

DSLIB - This names an overriding library containing either **SYM data** (within load modules) or **ADATA.** The library will be used by the **DMAP** command for building dsect maps. Up to 20 of these libraries may be defined. Example:
LITERAL DXDCUDSLIB1 C'DBCOLE.ZOSR21.DMAPS.FROM.COLESOFT'

To communicate these overrides to Z/XDC, these literals cause dump/XDC to generate and send to Z/XDC commands of the form: **DEFINE DSECTLIB <value>**

Warning: **DEFINE DSECTLIB** commands defeat the use of MAPLIBS by the **DMAP** command. Thus, when DXDCUDSLIB and DXDCUMAPLIB are both given:
- DXDCUMAPLIB will apply only to the **MAP** command, while
- DXDCUDSLIB will apply only to the **DMAP** command.

DWLIB - This names an overriding library containing **DWARF data** to be used by the **MAP** command for building source image maps for C and C++ programs. Up to 20 of these libraries may be defined.

To communicate these overrides to Z/XDC, these literals cause dump/XDC to generate and send to Z/XDC commands of the form: **LL REDIRECT DWARF PAT=**(*) USE=<value>(&2)**

CSLIB - This names an overriding library containing **C source** to be used by the **MAP** command for building source image maps for C and C++ programs. Up to 20 of these libraries may be defined.

To communicate these overrides to Z/XDC, these literals cause dump/XDC to generate and send to Z/XDC commands of the form: **LL REDIRECT CSOURCE PAT=**(*) USE=<dsn>(&2)**

How Do You Issue IPCS Commands?

In order to create IPCS literals, you need to issue IPCS's **LITERAL** command. So the question arises, how do you issue IPCS commands when using dump/XDC? Well, there are several ways:

- One good time to issue IPCS commands is prior to issuing the **VERBX XDCDUMP PRINT NOTERM NOTOC** command that starts up dump/XDC. See **HELP DUMPS STARTINGDUMPXDC** for information about starting dump/XDC.

- Or you can easily switch back and forth between Z/XDC and IPCS by using Z/XDC's **END** command to return to IPCS and (later) IPCS's **VERBX XDCDUMP PRINT NOTERM NOTOC** command to return to Z/XDC. See **HELP DUMPS STARTINGDUMPXDC SWITCHING** for more information about that.

- Or you can use Z/XDC's **IPCS** command to issue IPCS commands without leaving Z/XDC. See **HELP COMMANDS IPCS** for syntax and discussion.

Help DUmps MAppingdata Mmeffiles

Background

When debugging a dump originating from a foreign system, issues arise when it comes to obtaining mapping data matched to the programs, modules and data areas contained within the dump. **Module Mapping Extraction Files** contain mapping data extracted from load modules located at the foreign system. They are created by our **XDCMMEF** utility, which a third party customer can obtain from our website and run on his system.

For most mapping data pertaining to your program, it will already be at hand on your own system. But what is **not** always at hand is correct **ESD** data for building **Binder maps.** This is because when a load module gets relinkedited at the customer's site, the vagaries of the linkedit process can reshuffle the csects within the module.

The XDCMMEF Utility

So for Binder maps, we provide a utility to help with that. It is named **XDCMMEF.** It scans load libraries, extracts ESD data (and SYM data and ADATA(*), if any and if requested), compresses it and packages it for shipment. That file can then be sent, along with the dump, back to you for use during your dump/XDC session. It will enable Z/XDC to build maps that are correctly matched to the load modules and program objects contained in the dump.

For details about obtaining and running XDCMMEF, See **HELP UTILITIES XDCMMEF.**

(*) It is highly recommended that you **not** package ADATA into program objects. It should instead be packaged into ADATA side files. Including ADATA into program objects may create significant performance issues at bind time. See **HELP MAPS ADATA** for more information.

The Z/XDCMMEF Processing Overview

The point of the MMEF extraction file is to provide a way for your customers to send you load module mapping data without having to send you entire load modules (or program objects). This both reduces the amount of data that needs to be shipped and allows the customer to retain the privacy of his proprietary work.

In a single run, the XDCMMEF utility can extract mapping data for all load modules:
- Contained in your system's PLPA library list,
- Contained in your system's currently active linklist,
- Contained in up to fifty additional library concatenations.

For usage details, see **HELP UTILITIES XDCMMEF.**

In brief, the XDCMMEF utility works as follows:

- The utility can determine for itself the names and locations of all libraries comprising the PLPA library list.

- It also can determine the names and locations of all libraries comprising the linklist.

- In addition, the utility accepts as input up to fifty **LIBn** ddnames (//LIB1, //LIB2, ... //LIB50).

- Each **LIBn** ddname can reference a concatenation of any number (and any mix) of both PDS and PDSE load libraries.

- It is suggested that the **LIBn** ddnames match task library concatenations that may typically occur on the foreign system. Example: ISPF's ISPLLIB concatenation.

- For output, XDCMMEF writes a single, compressed, sequential file (RECFM=FB, LRECL=whatever) to //OUTFILE.

- This file is suitable for **shipment** as binary data over the Internet via FTP.

The MMEF Extraction File

Within the extraction file, the mapping data is organized as follows:

- For the **LIBn** Defined Libraries:

    - The extracted mapping data (ESD data, SYM data and ADATA) for a load module or program object is grouped under the **membername** of that module/object.

    - The load module mapping data for all load modules in a library is grouped under the **dsname** of that library.

    - The library's entire mapping data is grouped under the **LIBn** ddname to which

it was concatenated for the XDCMMEF processing run.

- For the **PLPA** Libraries:
  - The extracted mapping data (ESD data, SYM data) for a load module is grouped under the **membername** of that module.
  - The load module mapping data for all load modules is grouped under the category "PLPA modules".
  - There is no need to retain the names of the libraries that comprise the PLPA list.

- Ditto for the **Linklist** Libraries:
  - The extracted mapping data (ESD data, SYM data and ADATA) for a load module or program object is grouped under the **membername** of that module/object.
  - The load module mapping data for all load modules is grouped under the category "linklist modules".
  - There is no need to retain the names of the libraries that comprise the linklist.

Searching the MMEF Extraction File

All MMEF searches search for the mapping data pertaining to a given load module or program object. dump/XDC searches the MMEF file for the load module's extracted data in one or more of the following ways:

- When the search for a load module is governed by a **DDNTRANS** token, the search is restricted to the libraries grouped under the **LIBn** ddname referenced in the token. It is expected that such ddnames will correspond to task library concatenations. See **HELP COMMANDS DEFINE DDNTRANS** for information about these tokens.
- When searching for a load module known to be in either a PLPA list library or a linklist library, the search is restricted to only those load modules (and program objects) known to be in one or the other of those two library groups.
- Otherwise, when a module/object is known to be located within a given library (by dsname), that library's extracted mapping data is searched regardless of the **LIBn** ddname under which it is grouped.

Help DUmps MAppingdata DEfine

Background

The **DEFINE whatever** command is a new command added to Z/XDC for dump/XDC support. It is needed when dump/XDC is used to examine a dump that has been received from a downstream customer or other 3rd party data center. For more information, see **HELP DUMPS CONTENTS FOREIGNVLOCAL.**

The **DEFINE** command assists in the creation of various mapping library overrides that redirect the **MAP** and **DMAP** commands' searches for mapping data to look at libraries that are matched to the dump (instead of looking at the normal host system libraries).

dump/XDC allows you to supply special sources of information for dsects (the **DMAP** command) and load modules/program objects (mainly the **MAP** command), because this information will, in almost all cases, be different from the information used by default by Z/XDC.

Absent overrides, both the **MAP** and **DMAP** commands would hunt for mapping data using their normal search process:

- Checking MAPLIBS
- Checking host system task libraries
- Checking host system PLPA libraries
- Checking host system linklist libraries

But this doesn't work well when the necessary mapping data originates from a foreign system, such as might be the case when an ISV receives a dump from one of its customers. (See **HELP DUMPS EXPECTEDUSES ISVUSAGE.)**

Obtaining Mapping Data Matched to a Dump

When a dump is received from a third party (such as from a downstream customer), the necessary mapping data for the modules, control blocks and data areas from the dump is not in the dump. And for the most part, neither is it on the receiver's system. So where is it? Well, that depends...

z/OS System Control Block Dsects: In the case of dsects needed for mapping z/OS control blocks, if the dump originates from a downlevel system, then often the older versions of system control blocks are different (shorter, usually) then the current versions available on your local system. To address this issue, ColeSoft has available at its website (colesoft.com) mapping data for a couple of thousand z/OS cblocks for **each** level of z/OS ranging from current (R3.1) all the way back to **R1.6!** For more information, see **HELP DUMPS MAPPINGDATA ZOSCBLOCKS.**

Your Own Control Blocks: For your product's own private control blocks and data areas, you may wish to create locally on your own system historical archives of the necessary mapping data. See **HELP MAPS XDCMAPS YOURDOC** for suggestions.

Load Modules in the Dump: In the case of load modules (including program objects), yes. Load modules "can" contain SYM data and ADATA (for building csect and dsect

maps), but from a third party data center, they are not likely to.

- **But** all load modules will contain **ESD** data, and that will definitely be needed by Z/XDC for building **Binder** maps.

  However, to get the mapping data you need, it might seem like you would have to ask your customer to send you his load libraries, but of course that simply is not feasible for many reasons (size, privacy, ownership, trade secrets, etc.). So, we have created a better way...

- ColeSoft has written a utility named **XDCMMEF** which can scan entire load libraries and linklists, extract all mapping data from the load modules and program objects, compress it and package it for shipment via FTP. See **HELP UTILITIES XDCMMEF** for usage information.

- This extraction is called a Module Mapping Extraction File (an **MMEF** file). See **HELP DUMPS MAPPINGDATA MMEFFILES** for a discussion of its contents.

- When a downstream customer sends you a dump, he can also send you an MMEF file that is matched to that dump. You will then have in your possession all the data dump/XDC will need for building accurate maps of the load modules and program objects found within that dump. For more information, see **HELP DUMPS MAPPINGDATA.**

Setting Up Libraries of Dsect Mapping Data

- Once you have identified or obtained the necessary libraries of control block mapping data (SYM data or ADATA) matched to your dump, you can make them known to dump/XDC by naming them in a series of **DSECTLIB** tokens created by the **DEFINE DSECTLIB** command. See **HELP COMMANDS DEFINE DSECTLIB** for details.

- Alternatively, you can use IPCS' **LITERAL** command to set your dsect library names into a series of IPCS literals named **DXDCUDSLIBn.** See **HELP DUMPS MAPPINGDATA LITERALS** for more information about this.

Setting Up Libraries of Load Module Mapping Data

- When you have received an MMEF file, you can make its dsname known to dump/XDC via Z/XDC's **DEFINE MMEFDSN** command.

- Alternatively, you can use IPCS' **LITERAL** command to set your MMEF dsnames into a series of IPCS literals named **DXDCUMMEFDSNn.**

  Once MMEF datasets have been made known to dump/XDC, all I/O requests made by the **MAP** and **DMAP** commands (to any library whose extractions are contained within an MMEF file) will be intercepted and redirected to the MMEF file manager.

About MMEF Files and MMEFDSN Tokens

- When mapping load modules in an aspace's private area, the module will have been loaded into storage most likely either from a task library or from a linklist library. So unless an explicit override is provided on the **MAP** command, Z/XDC will

search first task libraries and then linklist libraries for the load module being mapped.

Absent dump related overrides, these searches will be conducted against local, host system libraries. However, when examining a dump from a foreign system, searching host system libraries is a pretty bad idea. This is where the **DEFINE MMEFDSN** command comes into play.

The **DEFINE MMEFDSN** command creates **MMEFDSN** tokens, each containing the dsname of an **MMEF** file. When these tokens exist, all accesses to load libraries by the **MAP** and **DMAP** commands are intercepted and redirected against the MMEF files named in the **MMEFDSN** tokens.

Searching MMEF Files by Library Dsname

(Note, additional information about the searches described here can be found in in **HELP COMMANDS DEFINE DDNTRANS MOREABOUT.)**

In brief, **MMEF** files contain extracted mapping data for load modules organized as follows:

- All load modules and program objects located in **linklist** libraries are identified as the linklist group.
- All load modules located in **PLPA** libraries are identified as the PLPA group.
- All libraries input'd to the **XDCMMEF** utility via **LIBn** ddnames are identified both via the **LIBn ddname** and via their **dsnames.**

See **HELP DUMPS MAPPINGDATA MMEFFILES** for more detailed information regarding the content of MMEF files.

So absent DDNTRANS tokens (see below), the mapping data search for private area load modules is conducted as follows:

- dump/XDC ascends the dumped aspace's TCB tree looking for OPEN'd task library concatenations.
- For each concatenated library, dump/XDC checks to see if the library exists in any LIBn section of the MMEF file.
- If so, then dump/XDC checks to see if that library contains an instance of the load module whose mapping data is being sought after.
- If so, then its ESD data is loaded, and a Binder map is built.
- If not, then dump/XDC moves on to the next ancestor's task library and repeats the cycle.
- If the mapping data is not found in any task library, then the linklist group is checked. This check is conducted without regard to any library's dsname.
- If the mapping data is not found in the linklist group, then the search is

continued into the next MMEF file (if any) listed in the next **MMEFDSNn** token or **DXDCUMMEFDSNn** IPCS literal.

- **MMEFDSNn** tokens are checked ahead of **DXDCUMMEFDSNn** IPCS literals.

Searching MMEF Files by Task Library Ddname

(Note, additional information about the searches described here can be found in in **HELP COMMANDS DEFINE DDNTRANS MOREABOUT.)**

When **DDNTRANS** tokens exist, the mapping data is searched for by task library ddname, not by load library dsname.

The **DEFINE DDNTRANS** command sets up correspondences between task library ddnames found in the dump and **LIBn** ddname groups found in MMEF files. So when **DDNTRANS** tokens exist, the mapping data search for private area load modules is conducted as follows:

- dump/XDC ascends the aspace's TCB tree looking for OPEN'd task library concatenations.
- If a **DDNTRANS** token exists that correlates that task library's ddname to a **LIBn** section of the MMEF file, then that section is searched without regard to any library's dsname.
- If mapping data for the desired load module is found, then its ESD data is loaded, and a Binder map is built.
- If not, then dump/XDC moves on to the next ancestor's task library and repeats the cycle.
- If the mapping data is not found in any task library's concatenation, then the linklist group is checked. This check is conducted without regard to any library's dsname.
- If the mapping data is not found in the linklist group, then the search is continued into the next MMEF file (if any) listed in the next **MMEFDSNn** token or **DXDCUMMEFDSNn** IPCS literal.
- **MMEFDSNn** tokens are checked ahead of **DXDCUMMEFDSNn** IPCS literals.

About the Preservation of Mapping Data from one dump/XDC Session to the Next

When you end your dump examination session, dump/XDC checkpoints all information pertaining to the maps you've loaded and the equates you've defined so that they can be automatically restored should you decide to resume your examinations at a later date. There are two mechanisms by which information is checkpointed and later restored:

- Information that is stored into **IPCS literals** is automatically saved by IPCS on a dump-by-dump basis, and is automatically restored when that dump is reOPEN'd at a later date. See **HELP MAPPINGDATA LITERALS** for more information about this.

- In addition, under the covers, dump/XDC automatically **journals** all commands and other information relating to the loading, setting and manipulating of maps, equates, display layouts, PF keys, and numerous other settings. When a session is ended, this journal is checkpointed into IPCS literals so that it can be preserved and played back when the dump is reOPEN'd at a later date. See **HELP MAPPINGDATA JOURNALING** for more information about this.

The DMAP Command's Search Process

For the **DMAP** command, during parse, Z/XDC extracts two pieces of information that are important: The name of the **dsect** map to be built, and the name of the **library member** to be searched. Notes:

- The name of the **library** itself usually is not given on the command, so **DMAP** itself will have to determine a set of libraries to be searched to find the given membername.

- The libraries to be searched will be either ADATA side file libraries or user and system load libraries.
  - In the case of load libraries, the given (or implied) membername will be a load module name.
  - In the case of ADATA side files (i.e. MAPLIBS), the membername can be unrelated to a load module name.

- In the case of ADATA side file libraries, the library names will come from the currently active **MAPLIBS** list.

- In the case of load libraries, the library will be a **task** library, an **LPALIB** library or a **linklist** library.

So for **DMAP** commands, the search proceeds as follows:

- First, the currently active **MAPLIBS** listed libraries are searched for ADATA. For each listed library, two members are search:
  - The member (if any) whose name matches the dsect to be mapped.
  - The member (if any) whose name matches that given (or implied) on the **DMAP** command.

- Second, if any **DSECTLIB** tokens or **DXDCUDSLIBn** literals exist, then the libraries named thereby are checked for the given membername.
  - If the member is found in an ADATA side file library, then that member is searched for ADATA belonging to the target dsect or csect name.
  - On the other hand, if the member is found in a load library, then:
    - The load module's ADATA (if any) is checked first,
    - Ahead of its SYM data (if any).
    - If the load library is also contained within an MMEF extraction file, then the search is diverted to the extraction file.
    - Otherwise, the load library needs to exist on the host system.

- If:
  - A specific load library is named by a **DMAP** command operand,
  - And if any **MMEFDSN** tokens or **DXDCUMMEFDSNn** literals exist,

  Then the named MMEF files are scanned. Then if the given library is found within

an MMEF file, and the desired dsect's mapping data is found within that library, then that data is used for satisfying the mapping request.

- The **USELDS** operand on the **DMAP** command can be used to force a specific dataset to be located on the host system, effectively ignoring the previous described override rules. dump/XDC assumes that you know what you are doing in this case.

The MAP Command's Search Process

The **MAP** command's search process is generally the same as for the **DMAP** command, except **DSECTLIB** tokens and **DXDCUDSLIBn** literals are not involved.

During parse, Z/XDC extracts two pieces of information that are important: The names of the **Binder and csect maps** to be built, and the name of the **library member** to be searched. Notes:

- The name of the **library** itself usually is not given on the command, so **MAP** itself will have to determine a set of libraries to be searched to find the given membername.

- The libraries to be searched will be either ADATA side file libraries or user and system load libraries.
  - In the case of load libraries:
    - The given (or implied) membername will be a load module name.
    - If a Binder map had not been previously built for the target load module, then searching a load library will be required.
  - In the case of ADATA side files (i.e. MAPLIBS), the membername can be unrelated to a load module name.

- In the case of ADATA side file libraries, the library names will come from the currently active **MAPLIBS** list.

- In the case of load libraries, the library will be a **task** library, an **LPALIB** library or a **linklist** library.

So for **MAP** commands, the search proceeds as follows:

- First, the currently active **MAPLIBS** listed libraries are searched for ADATA. For each listed library, two members are search:
  - The member (if any) whose name matches the dsect to be mapped.
  - The member (if any) whose name matches that which was given (or implied) on the **MAP** command.

- Second, if any **MMEFDSN** tokens or **DXDCUMMEFDSNn** literals exist, the identified MMEF files are searched to see if any of them contain mapping data for the load module to be mapped.

- The **USELDS** operand on the **MAP** command can be used to force a specific dataset to be located on the host system, effectively ignoring the previous described override rules. dump/XDC assumes that you know what you are doing in this case.

Help DUmps MAppingdata Journaling

Journaling is a mechanism by which dump/XDC records and preserves information about maps that have been built, equates that have been defined and other state data related to the dump/XDC session. This information is saved at the end of the dump/XDC session so that when the session is resumed later against the same dump, all maps, equates and other session state information can be automagically rebuilt.

In order to accomplish this restoration, dump/XDC journals the following Z/XDC commands for playback at the start of a resumed dump/XDC session:

- **DEFINE DDNTRANS**
- **DEFINE DSECTLIB**
- **DEFINE MMEFDSN**
- **DELETE DDNTRANS**
- **DELETE DSECTLIBS**
- **DELETE EQUATES**
- **DELETE MAPLIBS**
- **DELETE MAPS**
- **DELETE MMEFDSN**
- **DMAP**
- **DROP**
- **EQUATE**
- **LIBRARYLISTS (LL)**
- **MAP**
- **SET MAPLIBS**
- **SET QUALIFIER**
- **SET VDISPLAY**
- **SET VSETTINGS**
- **SET AUTOMAP**
- **USING**

Note, some commands generated by dump/XDC from IPCS literals are in the list above; however, commands generated by dump/XDC from IPCS literals are never journaled. (That would be redundant.)

Additional Information That Is Journaled

- Z/XDC's **CDF** (Current Display Pointer) and **NDP** (Next Display Pointer)

- All information that is a part of Z/XDC's **Session Profile.** This includes:
  - The **PF key** definitions
  - The **Watch Windows** layout
  - Numerous other controls that are managed via the **PROFILE** command.

  What is journaled is the **current** state of the profiled data. This includes all setting changes made since the Session Profile was loaded by Z/XDC's **PROFILE READ** (or RESET) command.

Under the Covers

When a dump/XDC session is ended properly (i.e. not aborted), dump/XDC preserves the journaled information by writing it to a series of **DXDCSwhatever** IPCS string literals. IPCS then saves its literals with the dump.

Later, when you wish to resume examining the dump, IPCS restores all literals. Then dump/XDC uses the restored data to restore maps, equates and other session state information. This is so that your resumption will go as smoothly as possible.

In order to reduce the number of IPCS literals needed, dump/XDC compresses the data before saving it. This compression can be turned off by creating a **//xxxNOSC DD DUMMY** allocation before starting your dump/XDC session.

Help DUmps MAppingdata Zoscblocks

z/OS control blocks tend to be quite stable across releases, but some do change. So when you receive and need to examine a dump from a system whose z/OS release level is different from your own, you may want to obtain mapping data for the level of z/OS control blocks that match those in the dump. Well, you can try to find and assemble the right level of mapping macros on your own system (assuming you still have the right level of z/OS macro libraries)... Or you could download the mapping data from us.

ColeSoft maintains at its website **release-specific collections** of IBM control block mapping data for all z/OS releases going back to **z/OS R1.6!** If you are both our customer and an IBM customer, then you can login to our website and download whichever collections of mapping data you need for your debugging.

Which Maps are Included

The z/OS mapping data consists of both ADATA and SYM data for a large number of z/OS cblocks segregated by z/OS release. The collections are named: **Z16MAPS,** ... **Z1DMAPS, Z21MAPS,** ... **Z24MAPS:**

- The oldest is for z/OS R1.6.
- The newest (as of this writing) is for z/OS R3.1.
- The z/OS releases 1.10 thru 1.13 are represented by Z1A, Z1B, Z1C and Z1D, respectively.

While the collections are extensive, they are not complete. In general, the following are true:

- They contain mapping data only for cblocks documented publicly by IBM's **z/OS MVS Data Areas** manuals (which can be found online by searching for **z/OS Internet Library).**
- They do **not** contain mapping data for OCO cblocks.
- They do not contain mapping data for every documented cblock.
- However, if mapping data for your favorite cblock is missing, let us know, and we will add it if we can.

How We Created the Mapping Data

There's no magic here. All we did was create assemblies of a large number of mapping macros, all available in **SYS1.MACLIB, SYS1.MODGEN** and a few other published macro libraries. We then captured and packaged the resulting **SYM data** and **ADATA** as follows:

- For the **SYM data,** we created a single load module named **ZnnMAPS.**

- For the **ADATA:**
  - We created a RECFM=VB library named **DBCOLE.XDC.ZnnMAPS.ADATA** containing 26 members.
  - Each member is named **ZnnMAPSa,** where **a** is a letter of the alphabet, A thru Z.
  - Each member contains ADATA for all cblocks whose dsect names start with the matching letter of the alphabet.
  - But be aware that this can lead to certain **unexpected outcomes.** Here's an example:
    - The mapping dsect for the Data Control Block (DCB) is named **IHADCB.**
    - So its mapping ADATA will be in member **ZnnMAPSI** (not ZnnMAPSD)!

The source code for these assemblies is already available at your Data Center in a Z/XDC distribution library whose factory default name is hlq.XDCV31.XDCASM. (Please check with your Systems Programmer for the library's local name.) If you're curious, feel free to take a look.

How We Packaged the Mapping Data for Download

First, we created a shipment library named **DBCOLE.ZnnMAPS.EXPORT.** It is an FB-80 PDS containing four members:

- **ZnnSYM** contains an XMIT of a load module named **ZnnMAPS** contained in a load library named **DBCOLE.XDC.ZnnMAPS.SYMDATA.** The load module contains no executable code, and is never intended to be LOAD'd into storage. Instead, it just contains **SYM** data for z/OS cblocks.

- **ZnnADATA** contains an XMIT of a RECFM=VB PDS library named **DBCOLE.XDC.ZnnMAPS.ADATA** containing 26 members named **ZnnMAPSA** thru **ZnnMAPSZ.** Each member contains the ADATA for z/OS cblocks whose dsect names start with the matching letter.

- **INSTALL** contains the JCL for a job that you can modify and run to install the mapping data files onto your system.

- **README** contains instructions for how to install the mapping data.

Second, we TERSEd the export library into a sequential, FB-1024 file named **DBCOLE.ZnnMAPS.TERSED** and uploaded it to our server.

How to Select, Download and Install a Mapping Data Collection

- Open a browser, and navigate to **colesoft.com.**
- Click on the **LOG IN** button (upper right), and Log in.

- Hover over **SUPPORT,** then over **Other Downloads,** then select **z/OS Mapping Collections.**
- Then select the z/OS release of interest.

Two items will be downloaded:
- A large, sequential FB-1024 file containing the TERSEd data described above.
- A small, sequential FB-80 file containing the JCL for the job needed to unTERSE and install the mapping data onto your system.

You will need to customize the job to your local needs and then run it.

Related Information

- The z/OS control block historic collections are similar to the **XDCMAPS** collection that we've been shipping with Z/XDC for decades. For a discussion of what **XDCMAPS** is and how to use it, see **HELP MAPS XDCMAPS.**
- For information about ColeSoft's online services, see **HELP SUPPORT ONLINE WEBSITE.**

Help DUmps Literals

IPCS has special equates called **literals** which contain string data rather than an address.

These literal equates can be divided into two groups:
1: Those used by dump/XDC. These are documented in **HELP DUMPS LITERALS DUMP/XDC**
2: Those not used by dump/XDC. These are documented in **HELP DUMPS LITERALS IPCS**

The following commands are available if dump/XDC is active to allow you to manipulate or use IPCS literal equates:
- The **DELETE ILIT** command.
- The **LIST ILIT** command.
- The **SET ILIT** command.
- The **ILIT()** builtin function in an address expression

Help DUmps Literals Dump/xdc

Typically, the process of troubleshooting a dump is a long, complex affair that often is conducted in several sessions over many days. So in order to preserve continuity from one session to the next, Z/XDC's state needs to be preserved on a dump-by-dump basis. This includes such things as:
- What mapping data libraries are connected to the dump,
- What maps have been loaded and the locations they have been assigned to,

- What Z/XDC equates have been created,
- What Z/XDC session profile is in use and what changes have been made to it,
- What changes have been made to the PF key definitions,
- What Watch Windows are currently defined on the display screen.

As noted above, all these things need to be preserved on a dump-by-dump basis. The best place to do that is in **IPCS string literals.** These are ideal because:

- They are easy to set up.
- The names and values of the literals are stored with the dump dataset and are made available to dump/XDC by IPCS **each time** a dump/XDC session is activated.
- The scope of a literal is a single dump, thus literals set for one dump have no effect upon any other dump.

As used here within Built-in Help, **IPCS string literals** are IPCS symbols assigned to represent character strings up to 256 characters long. dump/XDC uses those string values for setting and saving control information.

Throughout Built-in Help, these "IPCS string literals" will be referred to as simply **IPCS literals.**

How dump/XDC Uses Literals

dump/XDC uses IPCS literals in three ways:

1: Some IPCS literals are only set by the user, and are typically used to tailor dump/XDC processing.

2: Some IPCS literals can be set by either the user or Z/XDC. Such literals specify things that may be overridden by Z/XDC.

3: Some IPCS literals are only set by dump/XDC. In this case, the literal values are scrambled by dump/XDC, and other than deleting them when appropriate (via IPCS's **DROPSYM** subcommand), they should be left alone by the user.

Syntax

Literals are created by IPCS's **LITERAL** command and deleted by the **DROPSYM** command. When creating a literal for use by dump/XDC, use the following syntax:

LITERAL name C'value'

For dump/XDC, the layout of **name** needs to be:

DXDC<U><base-name><num-suffix>

Where:

DXDC - All literals related to dump/XDC start with the characters **DXDC** .

<U> - Literals that are user-alterable have a **U** here. Other literals used by dump/XDC will have some other character here.

<base-name> - This is the literal's base name. Literals having the same base name are related.

<num-suffix> - Some literals have a numeric suffix, indicating a collection of related values. When a suffix is required:
- It always starts at 1,
- It never has leading zeros,
- And the maximum permitted suffix varies according to what the base name is.

Deleting IPCS Literals

You can delete IPCS literals with IPCS's **DROPSYM** command. Here are some simple examples:

- **IPCS DROPSYM DXDCUJUNK** drops one literal named DXDCUJUNK.
- **IPCS DROPSYM (DXDCU:DXDCU)** drops all **user settable,** dump/XDC related, IPCS literals.
- **IPCS DROPSYM (DXDC:DXDC)** drops **all** dump/XDC related IPCS literals.

Displaying IPCS Literals

You can display the current values of literals with IPCS's **LISTSYM** command. Here are some simple examples:

- **IPCS LISTSYM DXDCUOPTIONS DISPLAY(STORAGE)** displays one literal named DXDCUOPTIONS.
- **IPCS LISTSYM (DXDCU:DXDCU) DISPLAY(STORAGE)** displays all **user settable,** dump/XDC related, IPCS literals.
- **IPCS LISTSYM (DXDC:DXDC) DISPLAY(STORAGE)** displays **all** dump/XDC related, IPCS literals.

Other than deletion (if required), you must not alter the values of any **DXDC** literals other than those where the names start with **DXDCU.**

User Settable Literals

The following are user settable literals supported by dump/XDC:

DXDCUCAS - This literal identifies the Current Address Space **(CAS)** (if any). See **HELP DUMPS EXECUTIONTHREAD** for more information.

DXDCUCSLIBn - This is a series of literals that link one or more C source libraries to the dump. These libraries will then be made available to Z/XDC's **MAP** command for building source image maps of C programs.

DXDCUCXU - This literal identifies the Current eXecution Unit **(CXU)** (if any). See **HELP DUMPS EXECUTIONTHREAD** for more information.

DXDCUDSLIBn - This is a series of literals that link one or more dsect libraries to the dump. These libraries will then be made available to Z/XDC's **DMAP** command for building dsect maps.

DXDCUDWLIBn - This is a series of literals that link one or more DWARF data libraries to the dump. These libraries will then be made available to Z/XDC's **MAP** command for building source image maps of C programs and for discovering the variables used by C programs.

DXDCULDDNXTn - This is a series of literals that link task library ddnames found within a dump to library definitions occurring within Module Mapping Extraction Files (MMEFs) received from the same customer that provided the dump. These will be needed by Z/XDC's **MAP** command when it has to build a Binder map (a map of where the csects are within a load module or program object).

Note, these DXDCULDDNXTn literals are not always necessary. See **HELP COMMANDS DEFINE DDNTRAN** for further discussion.

DXDCUMAPLIBn - This is a series of literals that name the MAPLIB libraries that are to be associated with this dump. These files will then be made available to Z/XDC's **MAP** and **DMAP** commands for building ADATA maps.

DXDCUMAXARBBU - This is a literal that overrides the default number of RBs examined if the **[.]ABBU** option is in effect. If not specified, or specified and the value is **NO** the default number examined is 4. The value specified must be a positive number in the range 2 to 255.

DXDCUMMEFDSNn - This is a series of literals that link one or more **Module Mapping Extraction Files** to the dump. These files will then be made available to Z/XDC's **MAP** command for building Binder maps. MMEF files are used by dump/XDC to help build Binder maps that match their corresponding load modules in a dump.

DXDCUOPTIONS - This literal contains a string of user selectable processing options that control various aspects of dump/XDC processing. See **HELP DUMPS STARTINGDUMPXDC PROCESSINGOPTIONS** for the details.

DXDCUPROFILE - This literal contains the name of the initial Z/XDC session profile that Z/XDC is to load at any start (both initial and resumption) of a dump/XDC session. See **HELP DUMPS MISCELLANEOUS PROFILES** for more information.

DXDCUPROFILE1 - This literal contains the name of the initial Z/XDC session profile that Z/XDC is to load at the initial start of (but not the resumption of) a dump/XDC session. See **HELP DUMPS MISCELLANEOUS PROFILES** for more information.

Help DUmps Literals Ipcs

Literal equates created internally by IPCS are documented in the **IPCS customization manual** (SA23-1383).

Help DUmps EXEcutionthread

What Was Running When the Dump Was Taken?

You'd think this would be an easy question, but it's not. In fact, there are dumps where it's really hard to figure out. In fact, there are dumps where it simply is not possible to figure out! It all depends.

What is the Current Execution Thread in a Running Program?

Baked into Z/XDC is a notion of something called a **Current Execution Thread.** When debugging a running program, this will be the environment in which the debugging target is executing. It includes:

- A current address space
- A current task (or SRB)
- A current RB (if not SRB)
- The current execution address
- The current registers
- A variety of current state information

All this is discussed in greater detail in **HELP EXECUTIONLEVELS.**

A considerable number of Z/XDC commands use this notion for choosing a default target for their actions. Examples:

- **LIST TASKS** displays the subtask tree for the current address space.

- **LIST RBS** displays the request blocks queue for the current task.

- **LIST RWREGS** displays the general registers from the current request block for the running program.

- **FORMAT R10?** displays the storage pointed to by the current value in R10.

In all cases, the commands can be targeted to **any** execution thread by the addition of suitable operands, but the **default** thread is set to whatever the current thread was when an ABEND occurred or a breakpoint was reached.

When debugging a running program, Z/XDC makes the determination of what the current execution thread is. But when debugging a dump, dump/XDC has to make that determination, and Z/XDC has to rely upon what dump/XDC tells it.

What is the Current Execution Thread in a Dump?

In order to inform Z/XDC what the Current Execution Thread is, dump/XDC uses two precursor concepts:

- **CXU:** This is the **Current eXecution Unit.** It identifies either a TCB/RB or an SRB from the dump under which execution is deemed to have been running.

  The CXU also identifies as current the address space in which the RB or SRB was running.

  The CXU can be set both automatically by dump/XDC and manually by:
  - The **SET CURRENT** command
  - The **CU** shortcut

- **CAS:** This is the **Current Address Space.** It identifies an address space as being current without also identifying a current RB or SRB.

- When the CXU is defined, the CAS is ignored.

The CXU and CAS are saved in the IPCS literals named **DXDCUCXU** and **DXDCUCAS,** respectively. They can be set:
- Either by IPCS's **LITERAL** command,
- Or by Z/XDC's **CU** shortcut,
- Or by Z/XDC's **SET CURRENT** command.

See **HELP DUMPS EXECUTIONTHREAD CHANGING** for information about managing the current execution thread settings that dump/XDC provides to Z/XDC.

At any point in time, dump/XDC will be in 1 of 3 modes with respect to what thread is current:
- Neither CAS nor CXU is set
- A CAS is set but not a CXU
- A CXU is set (so the CAS doesn't matter)

These modes and their consequences are explained in more detail below.

More About the CXU

dump/XDC's Current eXecution Unit describes quite a lot about an execution thread; not just the location of execution, but also what its registers and states are.

The location information in the CXU includes the following:
- The address space in which execution was occurring,
- The task or SRB under which execution was occurring,
- (For TCBs) the RB under which execution was occurring.
- The address at which execution was occurring.

The state data includes the following:
- The PIC (Program Interrupt Code)
- The ILC (Instruction Length Code)
- The PKM (key-mask)
- The BEA (Breaking Event Address)

- The FPC (Floating Point Control register)
- The HASID, PASID and SASID
- The general, access, and control registers
- The floating point and vector registers
- The guard registers (GSD GSSM and GSEPLA)
- The PSW (which includes):
    - The instruction address (IA)
    - The addressing mode (AMODE)
    - The address space control mode (ASC-MODE)
    - The condition code (CC)
    - The execution state and key

Much of this information is available from control blocks...
- That define an execution thread (ASCB TCB RB SRB XSB).
- That describe an ABEND (SDWA RTM1WA RTM2WA)
- That report CPU status (CPU status records)

But some of it is not, so it has to be either derived or done without:

- Control registers generally are not directly checkpointed in a dump, but much of their content can be derived by gathering bits and pieces from elsewhere (PASID for example, key-masks, feature control flags, etc.).

- CPU status records (available only in stand-alone dumps) do not record the PIC, ILC or BEA, so dump/XDC simply has to do without them.

- But CPU status records do contain a CPU's control registers, so those will be complete when presented to Z/XDC.

The CXU information that dump/XDC collects is passed to Z/XDC as error level state data. (See **HELP DUMPS EXECUTIONTHREAD ERRORANDRETRYLEVELS** for a discussion of what happens to error level and retry level execution environments in a dump/XDC session.)

Information that dump/XDC cannot collect is left zeroed in Z/XDC.

Determining Initial CAS and CXU Values

When dump/XDC first examines a dump, it will attempt to determine a current execution thread, and then set the **CAS** and **CXU** accordingly, resulting in one or another of the above three processing modes.

The determination process is strongly affected by the type of dump being processed (stand-alone dump vs. DUMP command dump vs. SDUMP[X] generated dump vs. whatever). For details about that decision process, see **HELP EXECUTIONTHREAD INITIAL.**

In many cases, however, the necessary information for making an accurate choice simply is not available in the dump, so dump/XDC frequently has to make **guesses** about what the CAS and CXU should be. They're good guesses, and they're often correct guesses, but when they aren't, commands and methods are available to you for overriding bad dump/XDC guesses. See **HELP EXECUTIONTHREAD CHANGING** for details.

Coping with the No CAS and No CXU Mode

When there is no current address space (CAS) and no current execution unit (CXU), then dump/XDC has been unable to identify a current execution thread. This can sometimes happen with certain dump types (for example, a stand-alone dump taken when a system falls into a disabled wait state).

In this no-CAS/CXU mode, many Z/XDC commands cannot be used without having to add operands because their default usages require the definition of a current execution thread.

But you can change that. You can override dump/XDC's CAS and CXU settings (or lack thereof) and set them to designate **any thread** you like as the "current" thread.

There are a couple of reasons why you might want to do this (even if you don't actually "need" to):

- If a dump's nature is such that a "current" execution thread simply does not exist, you can tell dump/XDC which thread you want to **pretend** is current.

- If dump/XDC has made an incorrect guess about what the current execution thread is, you can correct that choice.

- If you want to focus your attention on some task and RB other than the actual current one, then you can tell dump/XDC to **lie** to Z/XDC, thus relieving you of the need for appending overriding TCB address, RB address and/or ASID operands.

Again, see **HELP EXECUTIONTHREAD CHANGING** for how to change dump/XDC's **CAS** and **CXU** settings.

About the CAS but No CXU Mode

When a Current Address Space is defined but a Current eXecution Unit is not, Z/XDC commands that require a fully defined current execution thread will fail. However, commands that require only a current address space will not. Examples:

- **LIST RBS** fails.
- **LIST TASKS** succeeds.

When the CXU is Defined, the CAS Doesn't Matter

- If dump/XDC has been able to identify a specific execution unit as **current,**

- Or if the user has nominated a specific execution unit as 'current' (using the **CU** shortcut or other methods),

- Or if the **DXDCUCXU** literal has been set by IPCS's **LITERAL** command,

- Then dump/XDC will be able to tell Z/XDC proper what execution thread it is to consider as being current.

It does not really matter whether or not the selected execution unit really was current. It just gives dump/XDC something to tell Z/XDC so as to allow those Z/XDC commands to work that use a current execution thread as their default targets.

Z/XDC's current execution thread always requires that an **RB** (or SRB) be designated as being current. So, if (for example):

- You use the **CU** shortcut against a LIST TASKS display to designate some particular TCB as being current, then the task's newest RB will be regarded as current.

- **Exception:** If the newest RB is an SVRB for the **DUMP SVC** (SVC 51), then the next older RB will be deemed as being current.

When a current CXU is chosen or set, the current address space (CAS) setting is thereafter ignored.

Note that a CXU might not actually be "executing" at the moment the dump was taken. In fact, in most cases it will not be. But that doesn't really matter, does it?

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

INITIAL - Initial CAS and CXU determinations are affected by the type of dump being processed.

CHANGING - Reasons and ways to change the current execution thread that dump/XDC presents to Z/XDC.

ERRORANDRETRYLEVELS - The meaning of Error level and Retry Level under dump/XDC.

SRBMODE - When a dump's current execution thread is in SRB mode.

Help DUmps EXEcutionthread Initial

About the CXU and CAS Values

In order to inform Z/XDC what the Current Execution Thread is, dump/XDC uses two precursor concepts:

- **CXU:** This is the **Current eXecution Unit.** It identifies either a TCB/RB or an SRB from the dump under which execution is to be deemed to have been running.

  The CXU also identifies as current the address space in which the RB or SRB was running.

  The CXU can be set both automatically by dump/XDC and manually by:
  - The **SET CURRENT** command
  - The **CU** shortcut

- **CAS:** This is the **Current Address Space.** It identifies an address space as being current without also identifying a current RB or SRB.

- When the CXU is defined, the CAS is ignored.

The **CXU** and **CAS** values are maintained in the IPCS literals **DXDCUCXU** and **DXDCUCAS,** respectively. Being maintained by IPCS, the CXU and CAS values, once set, persist with the dump from one dump/XDC session to the next.

Whenever dump/XDC detects that the DXDCUCXU and DXDCUCAS literals are either **null** or set to **NO,** dump/XDC will analyze the dump to try and figure out (or at least make guesses about) which execution thread should be designated the **current thread.** This analysis is discussed below.

When dump/XDC opens a new dump, if the CXU and CAS fields are non-empty, then dump/XDC presumes that they were set or changed in a prior dump/XDC session, so it just leaves them as they are. If you wish to change these values, please see **HELP DUMPS EXECUTIONTHREAD CHANGING** for how to do that.

Figuring Out the Current Execution Thread (CXU)

As noted above, when dump/XDC begins processing a dump and the CXU and CAS are both not set, then dump/XDC tries to identify an execution thread in the dump that should be considered as being the **current** thread. Sometimes, that's doable, while other times it does not even make sense! Examples:

- If a dump was created by the System Operator's **DUMP** command, regardless of what address space(s) the Operator decides to dump, that space is never going to be "current". That's because technically, the current thread that is performing the dump is actually some SRB mode code running in the system's DUMP address space.

  Nevertheless, it would be a reasonable **guess** for dump/XDC to set the CAS and CXU as follows:
  - The Current Address Space **(CAS)** would be the one being dumped (or the first listed if several are being dumped).
  - The Current eXecution Unit **(CXU)** would be that aspace's jobstep task.

  Fortunately, if dump/XDC's guess is wrong, it is easily corrected by either the **SET CURRENT** command or the **CU** shortcut, and said correction will persist from one dump/XDC session to the next.

- If a dump is a **stand-alone** dump taken because the system went into a disabled wait state, then there is no guess that dump/XDC could make that would be reasonable.

So what happens when dump/XDC either doesn't make a guess or makes a wrong guess for the current execution thread? It's not a big problem. There are Z/XDC commands and IPCS methods you can use to override whatever setting dump/XDC did (or did not) make. See **HELP DUMPS EXECUTIONTHREAD CHANGING** for the details.

Informing Z/XDC What the Current Execution Thread is

There is no requirement that dump/XDC tell Z/XDC "the truth". Z/XDC doesn't really

care. All it will do is use the information that it receives from dump/XDC to set default targets for those commands that use currency information:

- **LIST TASKS**
- **LIST RBS**
- **LIST ESTAES**
- **LIST RWREGS**
- **LIST R10**
- **FORMAT R10?**
- And the like

Note, in all cases, the current execution settings only set **default** targets for these commands. All of them accept overrides that allow you to target any thread you like.

Initializing the CAS and CXU Settings

When dump/XDC first examines a dump, it attempts to determine a current execution thread, and then sets the **CAS** and **CXU** accordingly. The following describe the determination process for each dump type. As you will see, this process differs widely from one type to the next. (For information about supported dump types, see **HELP DUMPS DUMPTYPES.**)

In many cases, the **[.][NO]ABRB** and **[.][NO]FRRI** option settings have an effect on the chosen CXU. These options may be altered for a specific dump. See **HELP DUMPS STARTINGDUMPXDC PROCESSINGOPTIONS** for details on how to do that.

Initial CAS/CXU for Operator Dumps

When a dump is created by the **DUMP** Operator command, dump/XDC tries to work out the most likely address space or execution unit to choose as being current:

- The DUMP command requires the Operator to select one of more aspaces to be included in the dump.

- dump/XDC will pick the **first** aspace listed by the Operator, and set that as the **CAS.**

- If dump/XDC can access the job step TCB (via the ASXB), it will make that task and its newest RB the **CXU.**
  Based on the [.][NO]ABRB option the actual CXU may be altered.
  Based on the [.][NO]FRRI option the PSW and register information for the CXU may be altered.

- Otherwise, it will leave the CXU unset.

Each address space selected for dumping will be suspended during dumping. Thus, you will be able to locate all tasks and (suspended) SRBs that were running in that aspace.

Initial CAS/CXU for SYSMDUMPs

The Current eXecution Unit (CXU) will be set to the TCB/RB or SRB that ABENDed.
Based on the [.][NO]ABRB option the actual CXU may be altered.
Based on the [.][NO]FRRI option the PSW and register information for the CXU may be altered.

Initial CAS/CXU for SDUMPs

CAS and CXU determination depends upon whether the SDUMP[X] macro called dump services via an SVC or a branch entry.

SDUMPs - SVC Entry

This type of dump is always associated with task-mode work.

The Current eXecution Unit (CXU) will be set to the issuer of the SDUMP(X) macro, with the RB level associated with the caller of the SDUMP(X) macro (i.e. the task's second newest RB (if synchronous), or the task's newest RB (if asynchronous))
Based on the [.][NO]ABRB option the actual CXU may be altered.

There can never be an FRR active (having an FRR defined means that SVCs cannot be issued and an FRR can only be active if at least one FRR is defined) so the **[.][NO]FRRI** option does not apply.

SDUMPs - Branch Entry

This type of dump can be associated with:
- Normal task-mode work where the programmer has elected to use branch entry instead of SVC entry on the SDUMP[X] macro,
- A task with at least 1 FRR in effect, thus preventing the use of SVCs,
- SRB mode code.

Based on the [.][NO]ABRB option the actual CXU may be altered.

Based on the [.][NO]FRRI option the PSW and register information for the CXU may be altered.

Initial CAS/CXU for SLIP Dumps

If the dump was produced pursuant to a **SLIP** operator command, a further analysis is performed to determine the actual dump type. It can be any of the following:

- ABEND code SLIP dumps
- MSGID SLIP dumps
- ERROR EVENT SLIP dumps
- PER-type SLIP dumps. These are further divided:
    - Instruction fetch (IF)
    - Successful Branch (SBT)
    - Storage alteration (SA)
    - real Storage alteration (SAS)

- Zero-address detection (ZAD)

Once dump/XDC has classified the type, it will proceed to extract relevant information.

With all dumps, it is possible to use the **SET CURRENT** command (and **CU** shortcut) to change the CXU and CAS from their initial values determined by dump/XDC. However, even if they are changed, the **LIST DXDC SLIP** command will still display the original execution thread information.

SLIP Dumps - PER Type

dump/XDC will normally be able to identify the execution unit that caused the SLIP PER dump to be taken.

In addition, dump/XDC will be able (in most cases) to exactly identify the instruction that caused the SLIP PER dump.

Processing is as if the **[.]NOABRB** option is in effect.

Based on the [.][NO]FRRI option the PSW and register information for the CXU may be altered.

SLIP Dumps - ABEND Type

An ABEND type SLIP dump is a dump triggered by the occurrence of a specified ABEND completion code (specified via the SLIP command's **COMP=** operand).

dump/XDC normally can identify the execution unit that caused the SLIP ABEND dump to be taken. Additionally, dump/XDC can (in most cases) precisely identify the ABEND code and REASON code that caused the SLIP ABEND dump.

Based on the [.][NO]ABRB option the actual CXU may be altered.

Based on the [.][NO]FRRI option the PSW and register information for the CXU may be altered.

SLIP Dumps - MSGID Type

A MSGID type SLIP dump is a dump triggered when a message is WTO'd that has a specified message ID number (specified via the SLIP command's **MSGID=** operand).

dump/XDC normally can identify the execution unit from which the matched WTO was issued. Additionally, dump/XDC can (in most cases) precisely identify the message text that got matched.

Processing is as if the **[.]NOABRB** option is in effect.

Based on the [.][NO]FRRI option the PSW and register information for the CXU may be altered.

SLIP Dumps - ERROR EVENT Type

An ERROR EVENT SLIP trap is a trap that triggers on some event other than PER, ABEND or MSGID.

dump/XDC currently does nothing special with this type of SLIP dump. While it recognizes them, it does no special analysis. However, dump/XDC's normal analysis of a dump may be enough.

If dump/XDC chooses the CXU, processing is as if the **[.]NOABRB** option is in effect.

Based on the [.][NO]FRRI option the PSW and register information for the CXU may be altered.

Stand-Alone Dumps

Intrinsically, current execution information does not exist for stand-alone dumps, and cannot be reasonably guessed at. The user himself/herself must analyze the dump and then set CXU information. This analysis can be performed either via IPCS commands or within Z/XDC:

- If using Z/XDC commands, the **CU** shortcut command can be used to set the CAS and CXU information. See **HELP DUMPS SHORTCUTCMDS CU** for a discussion.

- The **SET CURRENT** command can also be used.

- If using IPCS commands, the **LITERAL** command can be used to change the value of the **DXDCUCXU** and **DXDCUCAS** literals. See **HELP DUMPS EXECUTIONTHREAD CHANGING** for more information.

Your settings will be saved by IPCS for later use should you choose to resume your dump/XDC session at a later time.

Help DUmps EXEcutionthread Changing

About the CXU and CAS Values

In order to inform Z/XDC what the Current Execution Thread is, dump/XDC uses two precursor concepts:

- **CXU** This is the **Current eXecution Unit.** It will identify either a TCB/RB or an SRB under which execution is to be deemed as running.

  It also identifies as current the address space in which the RB or SRB was running.

- **CAS** This is the **Current Address Space.** It identifies an address space as being current without also identifying a current TCB/RB or SRB.

- When the CXU is defined, the CAS is ignored.

CXU and CAS Persistence

The CXU and CAS values are maintained in the IPCS literals **DXDCUCXU** and **DXDCUCAS,** respectively. Being maintained by IPCS, the CXU and CAS values, once set, persist with the dump from one dump/XDC session to the next.

Whenever dump/XDC detects that the DXDCUCXU and DXDCUCAS literals are either **null** or set to **NO,** dump/XDC will analyze the dump to try and figure out (or at least make guesses about) which execution thread should be designated the **current thread.** This analysis is documented in **HELP DUMPS EXECUTIONTHREAD INITIAL.**

This detection occurs whenever a dump is first opened and again whenever the current execution thread is changed by Z/XDC's **CU** shortcut or **SET CURRENT** command.

Manually Changing the CXU and CAS Values

The CXU and CAS values can be set or changed:
- Either by IPCS's **LITERAL** command,
- Or by Z/XDC's **CU** shortcut,
- Or by Z/XDC's **SET CURRENT** command.

Using the CU shortcut

The **CU** shortcut can be used on any message relating to:
- An address space, an ASCB or an ASID
- A task or a TCB
- An RB
- Or an SRB

In addition, when the CAS or CXU have been overridden by the user, the **CU** shortcut can be used on the **LIST DXDC** command's report to reverse that overriding.

When used on a message that designates an address space, the **CU** shortcut clears the CXU and sets the CAS.

Otherwise, **CU** sets both the CXU and CAS (although the CAS does not matter when the CXU is set). This happens when **CU** is used on:
- A **LIST TASKS** display (The selected TCB's newest RB is designated as the current request block.)
- A **LIST RBS** display
- A **LIST SRBS** display

For more information, see **HELP SHORTCUTCMDS CU.**

Using the SET CURRENT Command

You can use Z/XDC's **SET CURRENT** command to set, clear or revert the CXU and CAS values as you see fit. When setting them, you can designate either an address space or a task or a request block or an SRB, again as you see fit. For details, please see **HELP COMMANDS SET CURRENT.**

Setting the CAS Using IPCS's LITERAL Command

To set or clear a Current Address Space **(CAS),** you can use the following IPCS command:

LITERAL DXDCUCAS C'value'

Where **value** (upcased for processing) is one of the following:

- **NO** causes the CAS to be not set. Example: **LITERAL DXDCUCAS C'NO'**
- Otherwise, anything that starts with an alphabetic (including the hex digits A-F) or national (@ # $) designates an address space matching that name. If multiple such (or no such) aspaces exist in the dump, an error is reported. Example: **LITERAL DXDCUCAS C'ABCD'**
- Anything that starts with a decimal digit (including 0) and consists entirely of hex digits and resolves to a nonzero value not wider than 2 bytes designates the address space matching that ASID. Said aspace must, of course, be in the dump. Example: **LITERAL DXDCUCAS C'0ABCD'**
- Anything else is an error. Example: **LITERAL DXDCUCAS C'1ABCD'**
- Notice, while IPCS requires the entire value string be enclosed within tic marks ('), dump/XDC does not require or even support that individual values within string be tic-enclosed. Thus, in the address space name example given above, the value string was given as C'ABCD', not as C''ABCD''.

Setting the CXU Using IPCS's LITERAL Command

To set (or clear) a Current eXecution Unit **(CXU),** you can use the following IPCS command: **LITERAL DXDCUCAS C'value[,value[,...]]'**

The individual sub-values can be hex numbers, character strings, dot-KEYWORDs or KEYWORD-equal-value values. Regardless, while IPCS requires that the entire value string always be enclosed by tic marks ('), dump/XDC neither requires nor permits individual sub-values to be tic mark enclosed.

The **values** allow you to nominate a CXU either implicitly or explicitly (or a bit of both).

It also allows you to control whether or not dump/XDC involves SDWAs when looking for an execution unit's current registers and PSW.

Many of the values are hex numbers. They can represent ASIDs, TCB addresses, SRB addresses or RB addresses. dump/XDC distinguishes their meanings by considering:

- Their positions in the values string relative to each other.

    - 1st, 2nd, 3rd

  - Their magnitude:
    - Greater or less than 64K
    - Greater or less than 16M
    - Positive or negative

  - The section of storage to which they resolve:
    - Above or below the line
    - Private storage vs SQA storage

Some of the values are dot-KEYWORDs or KEYWORD-equal-value values. They can occur anywhere relative both to each other and to the hex values. Note, the dot-KEYWORDs are called **control words.**

When mutually exclusive or contradictory values are given, an error is raised.

CXU Settings - Reverting

If you wish to revert the CXU to its default setting, then the value string must be either **NO** or **null.** Example: **LITERAL DXDCUCXU C'NO'**

CXU Settings - Selecting a Specific Execution Thread to be Current

You can setup the CXU to explicitly designate a specific execution thread as being current, but to do so, you will have to provide (or imply) two (or three) values in the DXDCUCXU literal:
  - The address space
  - The TCB or SSRB address
  - The RB address (for TCBs)

In some cases there are alternatives, but generally these three values can all be given as simple hex numbers. So, when parsing whatever numbers you provide, dump/XDC has to make several knowledge-based interpretations.

CXU Settings - Designating an Address Space

An address space can be specified by providing either its name (jobname, TSO userid or started task name) or its hex ASID number. So for a sub-value to be recognized as designating an address space, it must meet one or the other of the following criteria:

  - It is recognized as an address space **name** if:
    - It starts with an alphabetic (including A-F) or national (@ # $), and,
    - It consists entirely of alphabetics, nationals and decimal digits, and,
    - It is not longer than 8 characters, and,
    - It must not be tic mark enclosed.

    Example: **LITERAL DXDCUCXU C'ABCD,0123458'** This designates a job, TSO user or

started task named ABCD. Notes:
- This example further designates a TCB located at X'123458'.
- While the entire value string is tic mark enclosed, the individual values (in particular, the ABCD value) are not.

- A value is recognized as an **ASID** if:
  - It starts with a decimal digit (including 0), and,
  - It consists entirely of hex digits, and,
  - It resolves to a nonzero value, and,
  - The resolved value is 2 bytes wide or less. (i.e. from 1 to 65535)

  Example: **LITERAL DXDCUCXU C'0ABCD,0123458'** This designates an address space whose ASID is X'ABCD'. notes:
  - This example further designates a TCB located at X'123458'.
  - While the entire value string is tic mark enclosed, the individual values (in particular, the 0ABCD value) are not.

Of course, the designated address space must:
- Exist in the dump,
- Be unique in the dump (pertains to address space names).

CXU Settings - Designating a TCB

A task can be designated only by providing the hex address of a TCB within the designated aspace. In order to be recognized as a TCB address, it must:
- Start with a decimal digit (0 for example).
- Consist entirely of hex digits.
- Resolve to a value that is greater than 64K-1 (to distinguish it from ASIDs).
- Resolve to a value less than 16M (Because TCBs are always below the 16M line)
- Resolve to an address in the 24-bit private area or in the 24-bit system nucleus.

CXU Settings - Designating an RB

A request block can be designated by providing either the hex address of an RB chained from the designated TCB, or an RB's position on that chain, counting either from the chain's newest or oldest end. So, for a sub-value to be recognized as designating a Request Block, it must meet one or the other of the following criteria:

- A given value is recognized as an RB **address** if:
  - It starts with a decimal digit (0 for example), and,
  - It consists entirely of hex digits, and,
  - It resolves to a value that is greater than 64K-1 (to distinguish it from ASIDs), and,
  - It resolves to a value less than 16M. (RBs are below the 16M line), and,
  - It resolves to an address in the 24-bit private area.
  - It is given following a given TCB address.

- A given value is recognized as a position from the RB chain's **newest** end if:
  - It starts with a **minus sign,** and,
  - It is followed only by decimal digits, and,
  - It resolves to a value between -1 and -n (where n is the number of RBs present on the chain).

    - Note that **-1** is the **newest** RB on the RB chain (the same as not specifying an RB address or relative number at all), and -2 is next-newest RB, and so on.

  - A given value is recognized as a position from the RB chain's **oldest** end if:
    - It starts with a **hashtag (#),** and,
    - It is followed only by decimal digits, and,
    - It resolves to a value between #1 and #n (where n is the number of RBs present on the chain).

If a request block is not nominated using one of the above techniques, the most recently scheduled request block for the nominated TCB is used.

CXU Settings - locating an ABENDed RB.

If you wish to locate the most recent RB that has ABENDed, at or under the nominated (or defaulted) RB, use the **.ABRB** control word. Example:

LITERAL DXDCUCXU C'<ASID>,<TCB>,.ABRB'
This causes dump/XDC to locate the most recently abended RB for the TCB.

To **not** do this, use **.NOABRB** as the value string (this is the default).

Note:
- a maximum of 4 RBs under the nominated RB will be examined looking for an ABENDed RB.
- if an ABENDed RB is not found the nominated RB is used
- if an IRB is found the search is terminated.
- If the CXU is an SSRB or a CPU, the .[NO]ABRB control word is validated but ignored.
- if the **.[NO]ABRB** control word is not specified, the default is **.NOABRB** - not the current setting of the **ABRB** option.

CXU Settings - Designating SRB Mode Code

An SRB mode execution thread can be designated only by providing the hex address of an SSRB within the designated aspace. In order to be recognized as an SSRB address, it must:
- Start with a decimal digit (0 for example), and,
- Consist entirely of hex digits, and,
- Resolve to an address that is in common storage.

CXU Settings - CPU=hh

If a dump contains **CPU status** records, then you can use the **CPU=hh** keyword to set the CXU to whatever thread was being processed on the given CPU at the time of the dump.

The only dumps that contain CPU status records are **stand-alone dumps.** They are not present in any other kind of dump.

hh must be a 2-character hex value, and it must the id number of the desired CPU.

The CPU must be executing.

Unfortunately, not all state data is available for threads found via CPU status records. In particular, the records do not contain the ILC, PIC, or BEA. So this information will be missing when passed to Z/XDC.

For more information about what information is kept in the CXU, see **HELP DUMPS EXECUTIONTHREAD.**

CXU Settings - ABEND information

If the chosen CXU is an RB that has ABENDed (in this case there will be an ABEND SVRB scheduled over the top of it) or the chosen CXU is:
- the top RB for a task
- an SSRB
- executing on a CPU (stand-alone dumps only)

and dump/XDC detects that an FRR is active, the CXU will be marked as having ABENDed and the ABEND code and other information will be saved by dump/XDC for display by Z/XDC commands.

CXU Settings - using FRR information.

If an FRR (Functional Recovery Routine) is active, the current PSW and register values represent the current execution state within the FRR. However, it is most likely that the actual ABEND PSW and registers would be useful. The following control words can be used in the CXU to control this:
.FRRI - FRR SDWA information **is** to be used
.NOFRRI - FRR SDWA information **is not** to be used

If neither **.FRRI** nor **.NOFRRI** are specified, the setting of the [.][NO]FRRI option is used.

CXU Examples:

LITERAL DXDCUCXU C'CPU=05'
The execution unit currently running on CPU 05 will be made the current execution unit.

LITERAL DXDCUCXU C'ABCD,123458'
The Current eXecution unit is set to:
- The address space **named** 'ABCD',
- The TCB located at address 123458,
- Its newest RB.

LITERAL DXDCUCXU C'0ABCD,00123458'
Ditto except the selected address space will be the one whose **ASID** is X'ABCD'.

LITERAL DXDCUCXU C'0ABCD,00123458,-2'
Ditto except: The task's **2nd** newest RB will made current.

LITERAL DXDCUCXU C'0ABCD,00123458,#1'
Ditto except: The task's **oldest** RB will made current.

LITERAL DXDCUCXU C'NO'
The **DXDCUCXU** literal being set to **NO** will cause dump/XDC to revert the CXU to its default value for the dump.

Help DUmps EXEcutionthread Errorandretrylevels

About Error and Retry Execution Levels

When a program ABENDs and a recovery routine (ESTAE-type or FRR) receives control from the System's Recovery/Termination Manager (RTM), the RTM presents to the recovery routine information about **two** distinct execution environments:

- There is the execution's PSW, registers and state data that **did exist** at the instant the ABEND occurred. Z/XDC refers to this as the **error level** execution environment.

- There is the execution's PSW, registers and state data that **will exist** should program recovery be attempted. Z/XDC refers to this as the **retry level** execution environment.

For a more complete discussion, see **HELP EXECUTIONLEVELS.**

What About in Dumps?

However, it is important to understand that this distinction between error and retry level environments is an **artifact** created by the RTM. If Z/XDC receives control via a mechanism that **does not involve** an ABEND or program check, then the error level vs. retry level distinction collapses completely! There is no distinction. They are one and the same.

In the normal case, Z/XDC usually receives control pursuant to an ABEND or a breakpoint. In the dump/XDC case, however, Z/XDC receives control pursuant to a dump (not an ABEND or breakpoint). So consider:

- A dump generally will contain the frozen states of numerous execution threads...

- Of which any zero or more may be within various stages of RTM processing.

- Further, while the default current execution thread **(CXU)** might be an ABENDing thread, then again, it might not.

- Further, the user can use the **SET CURRENT** command (or **CU** shortcut) to change the designated CXU to anything he/she wants.

So Z/XDC's notions of error and retry levels must be adjusted a bit when running in a dump/XDC environment.

About Error and Retry Execution Levels in Dumps

Under dump/XDC, the CXU can be set to any thread the user chooses, so dump/XDC will tell Z/XDC that:

- The **error level** execution environment will be whatever the CXU is set to.

- The **retry level** execution environment will be identical to the error level environment, except for a minor adjustment to the PSW:

  - Typically, the error level PSW will point at, past or into the next machine instruction to be executed.

  - The retry level PSW, on the other hand, will be adjusted to always point at (not past or into) that instruction.

How does Z/XDC Use the Error and retry Environments?

These concepts of error and retry environments are deeply baked into Z/XDC.

In particular, Z/XDC uses the **retry level** state data to define and set **current** execution information, which is used to set default targets for a very large number of Z/XDC commands:

- SET ASID (defaults to the current home address space)
- LIST PSWE; FORMAT PSW? (defaults to the current PSW)
- LIST RWREGS/AREGS/VREGS/FREGS/etc; FORMAT R10? (defaults to the current register values)
- LIST TASKS (defaults to the current task)
- LIST RBS (defaults to the current request blocks)
- LIST ESTAES (defaults to the current recovery environment)
- LIST LSTACK (defaults to the current linkage stacks)
- WHERE (defaults to the current execution location)
- [And many more]

Z/XDC uses the **error level** state data to provide the targets for a number of commands specifically intended to display error level information:

- LIST EPSWE
- LIST ERWREGS
- LIST EAREGS
- LIST ER10; FORMAT ER10?

Under dump/XDC, the error level displaying commands will (except for the PSW) display the same information as the commands relying upon retry level data. (The

retry level PSW may have been changed, as described above, from the error level value.)

Help DUmps EXEcutionthread Srbmode

SRB mode execution is nearly invisible and often is ephemeral...

Unlike task mode execution (where TCBs exist for the life of a task), an SRB control block only exists to describe the **pendency** of an SRB routine, not the actual running of it!

Similarly, an SSRB (a cblock describing a suspended SRB) is created to checkpoint an SRB routine's state data when it is **suspended,** not when it is running.

In other words, when an SRB routine is actually executing at the time of a dump, it is quite possible that the cblocks that document the existence of the SRB routine might not, in fact, even exist!

So even if an SRB routine was running at the time a dump was taken, within the dump there may be no evidence whatsoever of that little factoid. Consequently, it is sometimes not possible for dump/XDC to show a dumped SRB routine's current PSW and registers even if the dump was taken while the SRB was actually running.

This is why SRB mode execution can be detected in only some dump types and not others. Specifically, dump/XDC can detect SRB mode execution in the following dump types:

- SLIP trap dumps where the code that triggers the trap is in SRB mode
- Branch entry type SDUMP[X] dumps where SDUMP[X]'s caller is running in SRB mode

In these cases, the CXU is set to represent an SRB mode execution thread.

In other cases, the following action is taken:

- SVC entry SDUMP[X] dumps - SRB mode cannot happen (SRB mode code cannot use any SVC except the ABEND SVC (SVC 13))
- Operator DUMP command - while under the covers the operator command uses a branch entry SDUMP[X] the issuing code is in TCB mode, not SRB mode.
- SYSMDUMP dumps - If there is no current TCB dump/XDC only sets the CAS if no user-specified CAS or CXU is specified.
- stand-alone dumps - dump/XDC never sets a CAS or CXU, so this is not an issue (also, see below)

In all cases, dump/XDC does not support allowing the **SET CURRENT** command to designate the SRB mode execution thread as the CXU. Only dump/XDC itself can do that.

Exception: For **stand-alone** dumps, the **SET CURRENT** command can designate as current any thread currently being executed on a specified CPU (Example: **SET CURRENT CPU=02).** It is quite possible for that CPU to be running an SRB mode thread.

Help DUmps COMmands

The following commands are only available when using dump/XDC:

- **DELETE ILIT** - delete an IPCS literal equate.
- **IPCS** - Issue an IPCS command etc.
- **LIST DXDC** - obtain information about a dump.
- **LIST DSTG** - obtain information about storage on a dump.
- **LIST ILIT** - obtain information about an IPCS literal equate.
- **SET CURRENT** - Set the current execution unit.
- **SET ILIT** - set the value of an IPCS literal equate.

The following commands produce different output when using dump/XDC:

- **LIST DSPACES** - List data spaces
- **LIST MSGS** - List messages about the current environment.

Help DUmps MIscellaneous

The following topics discuss in technical detail various aspects of actually using dump/XDC. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

- **DDnames** - Details about the DDnames dump/XDC recognizes and uses.
- **LOCKS** - Details about dump/XDC lock analysis.
- **PROFILES** - How you can use Z/XDC profiles with dump/XDC
- **CLOSED** - About the IPCS commands or command procedures executed by Z/XDC's **IPCS** command to close or change the current dump.
- **REDACTED** - How dump/XDC supports redacted dumps.
- **DATASPACES** - How dump/XDC supports data spaces.

Help DUmps MIscellaneous DDnames

DDnames

dump/XDC has the following modes of operation:

- As an IPCS verb exit program - before invoking Z/XDC to process the dump using XDC displays and commands.
- As Z/XDC while the user is processing the dump using XDC displays and commands.
- After returning from Z/XDC (when the session ends), before returning to IPCS.

In the first and last case, dump/XDC only recognizes a small number of ddnames itself. These ddnames are:

- xxxSDUMP
- xxxNOSC

In the middle case, when executing as Z/XDC, the ddnames are recognized by Z/XDC, and thus the 'standard' ddname documentation applies.

Help DUmps MIscellaneous Locks

dump/XDC provides 2 types of reports about locks:
- A report of the locks held by or requested by the Current eXecution Unit (CXU).
- A report of locks held across the entire system.

Reporting Locks Held by a CXU

The **LIST LOCKS** command (without operands) reports the locks held by the Current eXecution Unit. If there is no CXU, this command will display an error message.

The type of CXU will be displayed.

The locks held by or requested by the CXU will be displayed:
- If a suspend lock is held by the CXU this will be indicated.
- If a suspend lock has been requested by the CXU and not (yet) granted this will be indicated.
- If the CXU is not actually executing on a CPU then it cannot be holding the CPU lock or any spin locks or spinning on an unavailable spin lock - this will be indicated.
- If the CPU lock is held by the CXU this will be indicated (and the hold count will be shown).
- If a spin lock is held by the CXU this will be indicated.
- If the CXU is currently spinning on an unavailable spin lock this will be indicated.

Reporting Locks Held by the System

The **LIST LOCKS SYSTEM** command reports all locks held by all address spaces and CPUs in the entire system.

This command provides a complete report only for stand-alone dumps. If issued against any other type of dump, stand-alone dump, while it will work, the output of the command may not be complete.

The locks held (or, in some cases, attempted) will be displayed:
- If an address space's local lock is held as a local lock, this will be indicated.
- If an address space's local lock is held as a CML lock, this will be indicated. and the holding address space will also be indicated.
- If a CPUs CPU lock is held this will be indicated, along with the hold count.
- If a CPU holds one or more spin locks, this will be indicated, along with a list of their names.
- If a CPU is spinning on a spin lock, this will be indicated.

Help DUmps MIscellaneous Profiles

Information Persistence

One of the very nice features of dump/XDC is that all Z/XDC information pertaining to one dump:
- Is preserved with that dump from one dump/XDC session to the next,
- Does not affect similar information pertaining to a different dump.

The information that gets automatically saved includes:
- All maps (Binder, csect, dsect, SYM data, ADATA and C source maps)
- All equates
- All session profile settings. This includes:
    - The Watch Windows layouts
    - The function keys
    - Numerous session related settings and controls

dump/XDC saves the persistent information into each dump's index whenever you properly end a dump/XDC session. (A dump's index resides within the **userid.DDIR** dataset.)

Eventually, when you return to resume your examination of that same dump, the saved information is automatically reloaded. (If you wish to prevent this, you can use the **NORESTORE** exit parameter when you start the dump/XDC session. See **HELP DUMPS STARTINGDUMPXDC DXDCPROC** for more information.)

This reloading of dump related information occurs only when you **resume** a previously created examination of a given dump. It does not occur when you open up a dump for the first time.

Map and Equate Data

When you resume a previously opened dump/XDC session for a given dump, dump/XDC attempts to restore the maps and equates you had in place at the end of the prior session. But dump/XDC didn't actually save the prior maps themselves. Instead, it journaled the commands that created the maps and equates. So for the rebuild process, it just reissues those commands.

Session Profile Data

With respect to session profile data, when you open up a dump for the first time, there is no prior data, so Z/XDC is allowed to locate and load whatever session profile it chooses. See **HELP PROFILES DEFAULTPROFILE** for all the gory details.

On the other hand, when you **reopen** a previously examined dump, dump/XDC reloads the profile settings as they existed at the end of the prior dump/XDC session. Notes:

- The saved profile data was probably based on a session profile loaded by Z/XDC during a prior dump/XDC session.

- But the saved data will also include all changes (such as Watch Windows layouts, function key settings, and numerous other settings and controls) that may have been made during the prior dump/XDC session **even though** those changes might not have been saved by Z/XDC's PROFILE SAVE command!

- If this is not desired, you can always restore your preferred settings:
  - By using the **PROFILE READ** command to restore whatever session profile you like.
  - Or by using the **PROFILE(name)** argument on the **%DXDC** command used to start the dump/XDC session.

Overriding Session Profile Restoration Data

As noted above, when resuming a dump/XDC session against a dump, dump/XDC's default action is to restore session profile data that was saved in the dump's index when the prior dump/XDC session ended. There are several ways in which you can override this action, and cause dump/XDC to tell Z/XDC to load one of its session profiles from a profile library. They are as follows (listed in order of precedence).

- 1) You can add **PROFILE(name)** to the arguments on the **%DXDC** command used to start dump/XDC.

- 2) Otherwise, if this is the **first** dump/XDC session for a given dump, then dump/XDC checks for the existence of an IPCS literal named **DXDCUPROFILE1.** If it contains a valid value, then dump/XDC tells Z/XDC to search for and load that profile. This allows you to set an "initial profile" for a dump that can be subsequently altered and saved in the dump index and then reloaded for each subsequent dump/XDC session against this dump.

- 3) Otherwise, dump/XDC checks for the existence of an IPCS literal named **DXDCUPROFILE.** If it contains a valid value, then dump/XDC tells Z/XDC to search for and load that profile. (This is true regardless of whether this dump/XDC session is the initial session for the current dump or a resumption of a prior session.)

Notes:

- "Valid values" for profile names are names that are from 1 to 5 characters long consisting completely of alphabetic, numeric and national characters.

- If **both** DXDCUPROFILE and DXDCUPROFILE1 exist:
  - When opening a new dump for the **first** time, **DXDCUPROFILE1** will be used.
  - When reopening a dump to **resume** a dump/XDC session, **DXDCUPROFILE** will be used.

What About in the Batch?

When you start a dump/XDC session in batch, dump/XDC bypasses restoring prior session profile data from the dump's index. Likewise, when you end the session,

dump/XDC bypasses saving any profile data.

The restoration and saving of profile data is strictly reserved for interactive dump/XDC sessions. So batch sessions do not affect interactive sessions, and interactive sessions do not affect batch sessions.

You still, of course, may use Z/XDC's **PROFILE READ** and **SAVE** commands to load and save normal Z/XDC session profiles.

You may also use profile member name overrides when starting the dump/XDC session as described above in **Overriding Session Profile Restoration Data.**

Help DUmps MIscellaneous Closed

If dump/XDC detects that the IPCS commands and/or command procedures executed via the IPCS command has closed or changed the current dump, dump/XDC will terminate.

This is because the rug has effectively been pulled out from under dump/XDC, and dump/XDC is unable to continue.

Before termination, a message (DBC369) will be displayed, which requires an acknowledgement. When the acknowledgement is received dump/XDC will terminate.

Help DUmps MIscellaneous Redacted

Redacted Dumps

dump/XDC can process a 'redacted' dump.

A 'redacted' dump is a dump that may have one or more storage ranges 'redacted' (i.e. cleared to binary zero) to provide data privacy

The redaction process involves post-processing a dump to mark it as 'redacted' and to clear nominated storage ranges to binary 0.

dump/XDC processes redacted dumps as follows:
- The dump is handled 'normally' by dump/XDC
- the dump information section displayed by the 'LIST DXDC' command will indicate that the dump has been redacted
- The dataset information section displayed by the 'LIST DXDC' command will show how many redacted storage ranges were found
- When a storage area that has been redacted is displayed by the 'DISPLAY' or 'FORMAT' commands, an 'r' will be displayed as a prefix character indicating that some (or all) of the storage on that page may have been redacted.

Note that the architecture that IBM use to indicate redaction cannot indicate anything below the page level (4096 bytes) and it is impossible to distinguish

storage that has been redacted from storage in that page that had a value of binary zero.

Help DUmps MIscellaneous DAtaspaces

Data Spaces

dump/XDC can handle data spaces on a dump.

The data spaces actually present on the dump can be determined by using the dump/XDC version of the **LIST DSPACES** command.

A data space is accessed via an ALET. dump/XDC will correctly handle ALETs for the current execution unit.

However, there may not be a current execution unit (CXU), or even a current address space (CAS), or things may change (because the **SET CURRENT** command the **CU** shortcut has been used)

If required, (by use of the **SET ACCESSTO DATASPACE ...** command or the dump/XDC version of the **LIST DSPACES** command) dump/XDC allocates a **special** ALET for a given dataspace (identified by dataspace name and owning address space). The **special** ALET is distinct from any valid ALETs (in hardware terms) so can never be the same as any **real** ALET.

The **special** ALET will always refer to the nominated data space regardless of the CXU/CAS/none.

Help EQuates

Equates are individual labels that you can create and assign to represent locations in address spaces or data spaces.

- Equates are created by the **EQUATE** command.

- Equates can be deleted by the **DELETE EQUATE** command.

- Equates can be displayed by the **LIST EQUATE** command.

Equate labels must conform to the following rules of syntax:

- They may be from 1 to 64 characters long.
- Each character may be alphabetic, numeric, national (@ # $), or an underscore (_).
- Except the first character cannot be numeric.

Once an equate is created, you can use it in address expressions to refer to the storage location to which it has been assigned. Also, whenever the location containing the equate is displayed by the **FORMAT, SHOW** or **WHERE** command, Z/XDC will automatically show the equate label.

Floating Equates

Equates can be assigned to label either a specific location or a floating location. The location of "floating" equates can change over time according to the resolution of the address expression that defines the equate's location. Example:

EQUATE THISWQE .QHEAD?
EQUATE FIRSTWQE .QHEAD? F
Suppose that a work queue is anchored from a field named **QHEAD.** Then the first **EQUATE** command above assigns the label **THISWQE** to the work queue element that happens to be first on the queue at the time that the command is issued. Later, even when additional WQEs are added to the queue, or when that particular WQE is removed from the queue, the **THISWQE** equate will still label that specific WQE. This is because the first **EQUATE** command defines a "specifically located" equate label. The given address expression **(.QHEAD?)** is resolved by Z/XDC at the time that the **EQUATE** command is issued, and the resolved address value is assigned to the equate.

On the other hand, the second **EQUATE** command above defines a "floating" equate (because of the **F** operand). In this case, Z/XDC does **not** resolve the assigned address expression **(.QHEAD?).** Instead, the address expression itself is assigned to the equate label so that Z/XDC can re-resolve it every time the label is needed. Accordingly, **FIRSTWQE** will always label the first queued WQE no matter how often WQEs are added to or removed from the queue.

Note that if the above two **EQUATE** commands are issued at the same time, then initially they will probably label the same WQE, but as WQEs are added to or removed from the queue, the specific WQE labeled by **FIRSTWQE** will change, while the one labeled by **THISWQE** will not.

Area Equates

Equate labels can be assigned a length value. In other words, you can create an equate that represents an **area** of storage starting at a given address and continuing for a given number of bytes. Z/XDC makes use of equate length information in three ways:

- When storage is displayed, if the display starts within an area labeled by an equate, then the header of the display will include offset information relative to the starting point of the equate. Example:

  EQUATE BASE R7? 1000
  FORMAT .TOPLOOP
  Suppose that R7 is being used as a program base register. Suppose also that the label **TOPLOOP** is located in the program at +X'3D8' past the location pointed to by R7. Then the display produced by the **FORMAT** command will include **BASE+3D8** in its header line. (Note that the base address pointed to by R7 may **or may not** be the same as the actual start of program code.)

- When storage is displayed, if the storage being displayed falls within the reach of an area equate, and if the length of the equate's area is 32 bytes or longer, then the address of each display line will be shown as an offset from the start

of that equate.

- The length value of an equate can be used on the **FREEMAIN** command to control the amount of storage to be freed.

Display Formatting Controls

Equates can have attributes that affect whether Z/XDC's storage formatter displays storage as hex data or as machine instructions. Example:

```
FORMAT PGM+3D4
EQUATE OPENPLST PGM+3DC D
EQUATE OPENSVC OPENPLST+4 I
FORMAT OPENPLST-8
```

Suppose you want to use Z/XDC's **FORMAT** command to display a program that contains an embedded parameter list for the OPEN SVC. Suppose also, that a symbol map for this program is not available to Z/XDC. When maps are not available, then Z/XDC often has no reliable way to tell whether a given area of storage contains data or instructions, so it has to make assumptions that sometimes are wrong.

Specifically, in the absence of maps, the FORMAT command first assumes that the area to be displayed contains instructions. The command holds to this assumption until it encounters an invalid opcode; it then switches to data display mode, usually for the remainder of the display. Accordingly, in the above command sequence, the first **FORMAT** command will probably be thrown off and produce an incorrect display when it encounters the embedded plist.

The display errors can be corrected by use of the **EQUATE** command. The **D** in the first **EQUATE** command shown above forces Z/XDC's **FORMAT** command to switch to data display mode when it encounters the **OPENPLST** equate. The **I** on the second **EQUATE** command forces **FORMAT** to switch back to "instruction" mode when in encounters the **OPENSVC** equate. Accordingly, the second **FORMAT** command (above) will produce a more suitable display of storage than the first one did.

Autocloning

Multiple equates can be generated to represent each entry in a table or on a chain. The **EQUATE** command has operands that let you specify relevant information about the shape of a table entry or chain element (where the chain field is, for example), and the maximum number of entries to label. Each equate that is created shares a common root name followed by a unique sequence number.

More Information

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

BUILTIN - Built-in equates and other automatically defined equates.
FLOATING - Assigning fixed or floating bases to equates.

AUTOCLONING - Generating multiple equates to represent a table of entries or a chain of control blocks.

For additional information, see **HELP COMMANDS EQUATE.** [This topic will be displayed only if selected directly.]

Help EQuates Builtin

Equates are individual labels that represent locations in address spaces and data spaces. You can use the **EQUATE** command to create these labels, but Z/XDC itself provides a large number of these labels automatically. Some are "built-in" (that is, their definitions are hard-coded within Z/XDC), and others are automatically generated by use of various of Z/XDC's display and action commands.

The **built-in** equates provide labels and length information for a variety of important system structures, hardware items, and certain Z/XDC related internal objects. Examples include:

- Major areas of storage: The private areas, the SQA, the PLPA, etc.
- The save areas in which Z/XDC stores user program register sets: The general registers, the access registers, the control registers, the floating point registers, and the vector registers.
- The locations currently pointed to by the 32-bit and 64-bit views of the general registers.

Automatically generated equates are created by several of Z/XDC's commands to label the objects that those commands either display or create. Examples include:

- The **LIST TASKS** command generates **TCB#n** equates to label the actual Task Control Blocks whose information is being displayed.
- The **GETMAIN** command generates both an **AREA** equate and an **AREA#n** equate to label the area of storage that is obtained.

Subtopics Index

The various built-in equates and automatically generated equates are described in greater detail in the following subtopics. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

CBLOCKS - Built-in equates relating to certain z/OS control blocks.
STORAGE - Built-in equates that label major areas of storage.
REGSETS - Built-in equates relating to register sets.
REGISTERS - Built-in equates showing the storage pointed to by individual registers.
INTERNAL - Built-in equates relating to Z/XDC's internal structures.
OTHER - Built-in equates that label other locations of interest.
AUTOMATIC - Equates automatically generated by various display and action

commands.

You can use the **LIST EQUATES** command to view the complete set of (or subsets of) the equates (including built-in equates).

Help EQuates Builtin Cblocks

For by far the majority of z/OS control blocks, defining a built-in equate to represent them is not particularly needed. Z/XDC already offers plenty of ways for the user to create his own equates or to load dsect maps for those control blocks he might care about.

However, there are a few system control blocks for which the existence of a built-in equate provides significant benefit. They provide direct pointers to particular instances of certain debugging session related control blocks. Without these pointers, it would be difficult or impossible to create a generic address expression for finding these particular instances. Those equates are:

@ASCB - This points to the Address Space Control Block (ASCB) that controls the current Target Address Space. When the **SET ASID** command is used to put Z/XDC into Foreign Address Space Mode (FASM), this **@ASCB** equate will automatically change to point to the ASCB for the address space to which FASM has been set.

This is **extremely** useful! You can use the **EQUATE** and **USING** commands to create a whole structure of floating equates and maps that describe whatever structures within an address space that you want. Then if you float that structure off the **@ASCB** equate, then when you use the **SET ASID** command to switch from one address space to another, that entire structure of floating equates and maps will automagically move to represent the new target address space. For more information, see HELP COMMANDS USING.

RETRYRB - This gives the location and assumed length of the current retry level Request Block. The equate is reinitialized on every call to Z/XDC.

@SDWA - This gives the location and length of the current System Diagnostic Work Area (SDWA) representing the ABEND that caused Z/XDC to be called. The equate is reinitialized every time Z/XDC is called.

@SSCT - This gives the location and length of Z/XDC’s Subsystem Control Table (SSCT), if any. If Z/XDC’s SSCT does not yet exist, then this equate does not appear.

Help EQuates Builtin Storage

In the z/OS System, main storage is compartmentalized into several large areas of various types: 24-bit private storage, common storage, Pageable Link Pack Area, system queue area, system nucleus, etc. To aid in identifying (or referencing) such storage areas, Z/XDC has several built-in equates that identify their starts and lengths:

LOWCORE - This gives the starting address (0) and length (8K) of the common storage located below the private area. This area contains the System's PSA (Prefixed Storage Area).

PRIVATE - This gives the starting address and length of the 24-bit private area.

CSA - This gives the location and length of z/OS's 24-bit Common Services Area.

MLPA - If a 24-bit Modified Link Pack Area exists, then this gives its location and length.

FLPA - If a 24-bit Fixed Link Pack Area exists, then this gives its location and length.

PLPA - This gives the location and length of the 24-bit Pageable Link Pack Area.

SQA - This gives the location and length of the 24-bit System Queue Area.

XSQA - This gives the location and length of the 31-bit System Queue Area.

XPLPA - This gives the location and length of the 31-bit Pageable Link Pack Area.

XFLPA - If an 31-bit Fixed Link Pack Area exists, then this gives its location and length.

XMLPA - If an 31-bit Modified Link Pack Area exists, then this gives its location and length.

XCSA - This gives the location and length of z/OS's 31-bit Common Services Area.

XPRIVATE - This gives the location and length of the 31-bit private area.

JAVAWORK - This labels the 30-gigabyte **JAVA work area.** It starts at location 2G (X'80000000') and runs to location 32G.

GSYSAREA - This labels the 256-gigabyte above-the-bar **System Area.** It starts at location 32G and runs to location 288G.

GPVTLOW - This labels the above-the-bar **Low Private Area.** It follows the System Area and ends at the Common Area.

GGOMMON - This labels the above-the-bar **Common Area.** It follows the Low Private Area and ends at the Shared Area.

GSHARED - This labels the above-the-bar **Shared Area.** It follows the Common Area and ends at the High Private Area.

GPVTHIGH - This labels the above-the-bar **High Private Area.** It follows the Shared Area and ends at the top of virtual storage (X'FFFFFFFF_FFFFFFFF' i.e. at one less than 16 quintillion [a.k.a. "at 16 exabytes"]).

DEADZONE - (Deprecated) This labels the 2G line.

GAGALAND - (Deprecated) This labels the 4G line.

To view the complete set of built-in equates. use the **LIST EQUATES** command.

Help EQuates Builtin REGSets

Whenever Z/XDC receives control, one of the first things that it does is determines and saves the user program's registers into internal save areas. It saves both the general registers, the access registers, the control registers, the floating point registers and the vector registers (when they exist). Then whenever a command is issued that displays or uses any of these registers, Z/XDC gets the information to be displayed or used from these save areas.

For general registers and access registers, Z/XDC saves **two** complete sets, one for the error level environment, and one for the retry level environment.

Z/XDC has a complete set of built-in equates that identify the locations of these internal register save areas. These equates can be used in Z/XDC address expressions as an alternate way of accessing the data contained within the error level and retry level register sets.

The equates for these registers save areas are listed below. The registers are stored into these save areas in ascending numeric sequence (0 thru 15 for most register sets, 0 thru 31 for vector registers).

For certain wide registers (general registers and control registers), separated save areas are used for the high halves and the low halves. (This is why equates such as @RWREGS and @ERWREGS do not exist.) But for access registers, floating point registers and vector registers, the high and low halves are not separated.

RETRY LEVEL REGISTERS

@REGS - General registers, low halves (R0-R15)
@RHREGS - General registers, high halves (RH0-RH15)
@AREGS - Access registers, the entireties (AR0-AR15)

ERROR LEVEL REGISTERS

@EREGS - General registers, low halves (ER0-ER15)
@ERHREGS - General registers, high halves (ERH0-ERH15)
@EAREGS - Access registers, the entireties (EAR0-EAR15)

REGISTERS UNAFFECTED BY LEVEL

@CREGS - Control registers, low halves (CR0-CR15)
@CHREGS - Control registers, high halves (CRH0-CRH15)
@FREGS - Floating point registers, the entireties (FR0-FR15)
@VREGS - Vector registers, the entireties (VR0-VR31)

Z/XDC also has built-in equates that identify storage pointed to by individual registers. See HELP EQUATES BUILTIN REGISTERS for more information.

To view the complete set of built-in equates, use the **LIST EQUATES** command.

Help EQuates Builtin REGIsters

When registers point to storage, and you are viewing storage near where a register is pointing, it is useful to see (a) that a register is pointing to a nearby location, and (b) how far away that nearby location is.

In order to display this kind of information, Z/XDC provides built-in equates the are associated with each register in the several register sets. Each such equate:

- Labels the storage being pointed by a register,
- Has a name based on the name of the register with which it is associated, and
- Has a length attribute **(4K)** so that an offset from the pointed-to location will be shown for all storage displays that are at or within **4K** following that location.

The location labeled by the equate changes, of course, whenever the contents of the register to which it is associated changes.

The Sets of Register Equates

The following built-in register equates exist:

- **@Rn** (@R0, @R1, ..., @R15)
  These label the locations pointed to by each of the sixteen 32-bit retry level general registers.

- **@RWn**
  These label the locations pointed to by the sixteen 64-bit retry level general registers. **BUT** an individual **@RWn** equate will not exist if the hi-order word of the 64-bit register is either all **00s** or all **FFs:**

  - When the hi-order word is **00000000,** @RWn points to the same location as @Rn, so it is not needed, and if it were to exist it would serve only to clutter up FORMAT'd displays unnecessarily.

  - When the hi-order word is **FFFFFFFF,** that's a common indication that the high half of the register has not yet been used or set. In any case, defining the equate would place it somewhere way up within the last four gigs of addressing's **sixteenth exabyte** (currently an absurd result)!

- **@ERn**
  These label the locations pointed to by each of the sixteen 32-bit error level general registers. **BUT** at any given point in time, an individual **@ERn** equate may or may actually exists. Whether or not it does is governed by the following rules:

  - If the retry level and error level environments are different, then all **@ERn** equates will exist.

  - Otherwise, if a given register's, error level and retry level values are

different, then that particular **@ERn** equate will exist. (Note, this can happen when the **ZAP** command has been used to change the contents of a retry level general register.)

- **@ERWn**
  These label the locations pointed to by each of the sixteen 64-bit error level general registers. **BUT** at any given point in time, an individual **@ERWn** equate may or may actually exists. Whether or not it does is governed by the following rules:

  - If the hi-order word of the 64-bit register is all **00s,** then the **@ERWn** equate will **not** exist. (In other words, when an ERWn register points to the **same location** as the same-numbered ERn register, then the **@ERWn** equate is suppressed.)

  - If the hi-order word of the 64-bit register is all **FFs,** then the **@ERWn** equate will **not** exist. (Being all FFs often is an indication that the register's high half has not yet been used or set.)

  - For all remaining **@ERWn** equates, if the retry level and error level environments are different, then those **@ERWn** equates will exist.

  - Otherwise, if a given wide register's, error level and retry level values are different (in spite of the environments being the same), then that particular **@ERWn** equate will exist. (Note, this can happen when the **ZAP** command has been used to change the contents of a retry level general register.)

- **@RHn** and **@ERHn** equates do not exist because by themselves they generally do not contain storage pointers.

Error Level vs. Retry Level Registers

Generally, there are two conditions under which the error level and retry level registers will differ:

- When the error level and retry level environments are different. (See **HELP EXECUTIONLEVELS** for more information.)

- When the **ZAP** command has been used to change the contents of one or more of the retry level registers. (The error level registers cannot be zapped.)

General Benefits

These register target equates have the following features and benefits:

- **All** of the general registers are considered to be pointers. (Whether or not a particular registers actually does contain a pointer and not just miscellaneous data is of no particular importance.)

- The register equates are assigned a length of 4K. This means that if storage is displayed at or within 4K of the location pointed by the register, the offset of that storage from that equate will be shown. In other words, if displayed

storage is classically addressable by a register, then that register's equate name will be included in the display. (Note, the presence or absence of the Long Displacement Facility is not considered.)

- The equates are assigned the **GLOBAL** attribute, so storage that is displayed from any address space, from any data space, and even from real storage will be labeled by those equates whose addresses fall at or near the displayed storage.

Z/XDC also has built-in equates for Z/XDC's internal save areas for all of the user program's error level and retry level register sets. See **HELP EQUATES BUILTIN REGSETS** for more information.

To view the complete set of built-in equates, use the **LIST EQUATES** command.

Help EQuates Builtin Internal

The following built-in equates related to Z/XDC's internal information. A couple may be of use to customers, but most will not.

@XDC - This equate labels the load point and length of the **xxx** load module. This may be useful to some customers. **("xxx"** usually is **"XDC",** but see HELP XDCCLONES regarding changing **XDC's** name to something else.)

@XDCEP - This equate identifies the **xxx** load module's entry point. This may be useful to some customers.

@DBC - This is an alias of the **@XDC** equate. **("DBC"** was Z/XDC's original name, until 1984 or so.)

@LOCAL - This gives the location and length of Z/XDC's temporary, per-call, control blocks area. This area starts with Z/XDC's LCLBLOCK and continues with various other control blocks, all of which are released upon every return to the user's program and rebuilt upon every call back into Z/XDC. (This area is internal to Z/XDC.)

@GLOBAL - This gives the location and length of Z/XDC's global data area (a data area internal to Z/XDC).

@SSCT - This gives the location and length of Z/XDC's Subsystem Control Table (SSCT) if any. If Z/XDC's SSCT does not yet exist, then this equate does not appear.

Help EQuates Builtin Other

Here are a few other equates that don't fall into other categories but still are of general interest.

@BEA - This labels the retry BEA (Breaking Event Address) address for your program. This is the same address displayed by the **LIST BEA** command (with no RB or SSRB address).

The **Breaking Event Address** is IBM-speak for the most recently executed branch-type instruction that (in this case) was executed prior to the ABEND or breakpoint by which Z/XDC received control. **Branch-type instruction** is Dave Cole speak for any instruction that can change the execution path. This includes branching instructions, jumping instructions, SVCs, PCs, LPSWs and many others.

Sometimes, the BEA will point to a **LPSW** instruction somewhere in the System Nucleus (which is kinda useless for debugging purposes), but usually it will point to a branch-type instruction within your own code. So the **@BEA** equate and the **LIST BEA** command can be very helpful if you need to know how you got to where you are.

WARNING! The hardware support for the **BEA** only records the address. It does not record the address space in which that address is located. Consequently, under certain conditions:

- The location displayed by the **LIST BEA** command might be in the wrong address space.
- The location assigned to this **@BEA** equate might be in the wrong address space.

This situation can occur when the last prior flow-of-execution changing event (branching instruction, PC instruction, whatever) occurred in one address space while the interrupt by which Z/XDC received control occurred in a different address space.

The best example of this occurs when a trap is placed at the start of a space switching PC routine. When Z/XDC receives control due to the trap, the execution address space will be that in which the PC routine resides. But the BEA location will be at the PC instruction located in the calling address space.

Note that the **#BEA** equate may be created by the **LIST BEA** command under some circumstances. The **#BEA** and **@BEA** equates are independent of each other and may or may not point to the same location.

@CDP - This shows the location of the **Current Display Pointer.** The **CDP** generally references:

- The most recently displayed location in storage (when set by the **DISPLAY** and **FORMAT** commands)

- The location most recently referenced by the **VERIFY** command (for use by an immediately following ZAP command)
- The most recently zapped location in storage (when set by the **ZAP** command)
- The most recently displayed retry level execution address (when set by the **WHERE** command)
- The most recently described location (when set by the **LIST SUBPOOLS anyaddress** command)
- The most recently found data (when set by the **FIND** command)
- etc. (see HELP ADDRESSING IMPLICIT CDPNDP for a complete list of commands that set the **CDP.)**

IMPORTANT! Each Watch Window in a full screen display has its own **CDP** value. So a **CDP** location seen within one window generally will not be the same as a **CDP** location seen in another window!

@EBEA - This labels the error BEA (Breaking Event Address) address for your program. This is the same address displayed by the **LIST EBEA** command.

This equate only exists if the error level and retry level is different.

The **Breaking Event Address** is IBM-speak for the most recently executed branch-type instruction that (in this case) was executed prior to the ABEND or breakpoint by which Z/XDC received control. **Branch-type instruction** is Dave Cole speak for any instruction that can change the execution path. This includes branching instructions, jumping instructions, SVCs, PCs, LPSWs and many others.

Sometimes, the BEA will point to a **LPSW** instruction somewhere in the System Nucleus (which is kinda useless for debugging purposes), but usually it will point to a branch-type instruction within your own code. So the **@EBEA** equate and the **LIST EBEA** command can be very helpful if you need to know how you got to where you are.

WARNING! The hardware support for the **BEA** only records the address. It does not record the address space in which that address is located. Consequently, under certain conditions:

- The location displayed by the **LIST EBEA** command might be in the wrong address space.
- The location assigned to this **@EBEA** equate might be in the wrong address space.

This situation can occur when the last prior flow-of-execution changing event (branching instruction, PC instruction, whatever) occurred in one address space while the interrupt by which Z/XDC received control occurred in a different address space.

The best example of this occurs when a trap is placed at the start of a space switching PC routine. When Z/XDC receives control due to the trap, the execution address space will be that in which the PC routine resides. But the BEA location will be at the PC instruction located in the calling address space.

- **@PRIOR** - This displays the retry level resume address that existed the **prior time** that Z/XDC received control from the user program. This is the machine instruction whose operands are displayed by a **LIST OPERANDS PRIOR** command.

- **@LEPILOG** - This represents the current Language Environment epilog address. This is the location that the current function will branch to when it executes a return statement or reaches the end of the current function definition. This is commonly shown as a result of the **[E]WHERE** command.

- **@TEA** - This displays the storage violation address known as the translation exception address (TEA), with the last 12 bits zero'd. When flag SDWATEAV indicates that the SDWATRNE contains an address, @TEA will contain that address. Otherwise @TEA will be zero. The last 12 bits are zero'd so that commands such as **DISPLAY** and **FORMAT** will function accurately when using @TEA in an address expression. For example, you may see that the high half of the TEA is nonzero because it wasn't cleared as intended while in 64-bit mode. If it were 0000010A, you could issue **D @TEA-10A00000000** to easily see whether the address would be correct if the high half were cleared. Note that if you issue **D @TEA** or **F @TEA,** z/XDC might show you allocated storage. In that case, z/XDC itself requested storage and z/OS happened to allocate what @TEA points to. However **D @TEA** and **F @TEA** will often encounter the same translation exception as the program being debugged. Examples of storage violations known as translation exceptions include the page translation exception (ABEND 0C4-10) and the segment translation exception (ABEND 0C4-11).

Help EQuates Builtin Automatic

Many of Z/XDC's display and action commands, as a side effect, automatically create equates to label the objects that the command might be displaying or creating. Usually, multiple equates will be created. And usually, the equates will be numbered from #1 upwards.

When creating a numbered series of equates, the objects being labeled will, when it makes sense to do so, be labeled from oldest to newest with **equate#1** labeling the oldest instance. Z/XDC will do this even if it has to label a queue backwards in order to do so. Examples: request block (RB) queues and STAE control block (SCB) queues.

- Before creating a series of equates, if a prior series of similar equates already exists, then Z/XDC will always purge the entirety of the prior series before creating the new series. So if you want to preserve labels for one or more prior instances, you will first have to use the **EQUATE** command to create your own names for those equates prior to using the command that will recreate the automatic equates. Example: **EQ THISRB RB#2;L RBS**

Warning! All of the following automatic equates are **static!** They do **not** float. Their

locations will change (if needed) **only** when the command that generated them is reissued. This means that these equates can become **obsolete** without warning should the data upon which they are based change. For some of these equates, such changes are quite likely. For others, not so much.

The following automatic equates will be generated by the following commands:

ALE#n - This is a series of equates giving the locations and lengths of each of the Access List Entries that form an Access List being displayed by the **LIST ACCESSLISTS** command.

APE#n - These equates will be generated by the **LIST PGMS** command when it displays information about load modules loaded by CICS's internal services. They give the locations and lengths of the Application Program Elements that CICS uses to describe load modules that it has brought into storage via its own services (not via z/OS's LOAD service).

AREA - This equate is created by Z/XDC's **GETMAIN** command. It always points to and has the same length as the area of storage **most recently** obtained by the command.

AREA#n - This is a series of equates giving the locations and lengths of each area of storage obtained by Z/XDC's **GETMAIN** command. A new equate is created each time the command is used. (Prior **AREA#n** equates are **not** purged.) **n** is an automatically incrementing identifier.

#AREGS - This equate gives the location and length of the system control block field in which the **LIST AREGS** command found the access register data that it displayed. This equate is regenerated each time the **LIST AREGS** command is issued with an **rbaddress** or **ssrbaddress** operand. It is **not** regenerated when the LIST AREGS is given **without** operands.

ASTE#n - This is a series of equates giving the locations and lengths of **ASN Second Table Entries.** ASTEs are used by the System to define both address spaces and dataspaces and to anchor their page tables. **ASTE#n** equates are (re)created by two commands:

- The **LIST DSPACES** command displays information about dataspaces owned by a given address space. In this case, the **ASTE#n** equates are generated in alphabetic order by dataspace name.

- The **LIST ACCESSLISTS** command displays information about address spaces and dataspaces pointed by Access Lists. In this case, the **ASTE#n** equates are generated in ALE sequence.

CDE#n - These equates will be generated by the **LIST PGMS** command when it displays information about load modules located in private storage on Job Pack Queues (JPA) or in common storage on the Link Pack Queue (LPA, DLPA, MLPA or FLPA, but **not** PLPA). They give the locations and lengths of the Content Directory Entries that z/OS uses to describe most load modules in storage.

#dspacename - These are equates that are created by either the **LIST ACCESSLISTS** command or the **SET ACCESSTO DATASPACE** command.

- In the case of the **LIST ACCESSLISTS** command, the equates are

created only for those dataspaces that are accessible to (ie, can be displayed by) Z/XDC.

- In the case of the **SET ACCESSTO DATASPACE** command, an equate is created for the dataspace to which access is established.

For each dataspace, these equates label their **X'00000000'** location. Accordingly, these equates provide an easy way to reference locations within the dataspace **without** having to use either an access register or the ~ALET() built-in function.

FAKE#n - These equates will be generated by the **LIST PGMS** command when it displays information about Privately Loaded load modules whose existence has been established via special usage of the **MAP** and **DMAP/USING** commands. These modules are known only to Z/XDC, not to the rest of the System. These equates provide the locations and lengths of fake LPDEs that Z/XDC has constructed to describe these modules.

GUARD#n - This is a series of equates giving the locations and lengths of the guard areas (if any) located within above-the-bar memory objects for a given address space. These equates are recreated each time the **LIST MEMORYOBJECTS** command is used to display a summary of an address space's memory objects. **n** corresponds to the sequence numbers shown in the **LIST MEMORYOBJECTS** display.

JDE#n - These equates will be generated by the **LIST PGMS** command when it displays information about load modules loaded by JES3's internal services. They give the locations and lengths of the JES3 Directory Elements that JES3 uses to describe load modules that it has brought into storage via its own services (not via z/OS's LOAD service).

LPDE#n - These equates will be generated by the **LIST PGMS** command when it displays information about load modules located in The System's Pageable Link Pack Area (PLPA). They give the locations and lengths of the Link Pack Directory Entries that z/OS uses to describe all modules located in the PLPA.

Warning! A LIST PGMS PLPA command will generate **thousands** of these equates. If that causes Z/XDC to be sluggish, a **DELETE EQUATES LPDE#*** command will get rid of them.

LSE#n - This is a series of equates giving the locations and lengths of each state type Linkage Stack Entry queued for a selected task or SRB routine in an accessible address space. These equates are recreated each time the **LIST LSTACK** command is used to display a summary of a linkage stack. **n** corresponds to the sequence numbers shown in the **LIST LSTACK** display.

LSE#0 - An **LSE#0** equate is also generated to label a linkage stack's first header entry. Thus, even the display of empty stacks will cause at least one equate to be generated. (Equates are **not** generated for 2nd and subsequent header entries. They also are not generated for trailer entries.)

MOB#n - This is a series of equates giving the locations and lengths of the above-the-bar memory objects for a given address space. These equates

are recreated each time the **LIST MEMORYOBJECTS** command is used to display a summary of an address space's memory objects. **n** corresponds to the sequence numbers shown in the **LIST MEMORYOBJECTS** display.

MOBDATA#n - This is a series of equates giving the locations and lengths of the unguarded data areas located within the above-the-bar memory objects for a given address space. These equates are recreated each time the **LIST MEMORYOBJECTS** command is used to display a summary of an address space's memory objects. **n** corresponds to the sequence numbers shown in the **LIST MEMORYOBJECTS** display.

OTCB#n - This is a series of equates giving the locations and lengths of each Openy Task Control Block in the Target Address Space. An OTCB only exists if this is a USS (Unix System Services) task or if the task has issued a USS request. These equates are recreated each time the **LIST TASKS** command is used. **n** corresponds to sequence numbers shown in the **LIST TASKS** display.

#BEA A - This equate is regenerated every time the **LIST BEA rborssrbaddress** command is issued wherein an **rborssrbaddress** operand is explicitly given. (It is not regenerated when the LIST BEA command is given without operands.)

This equate labels the breaking event address (BEA) that is stored when the RB or SRB is interrupted.

#PSWNIA - This equate is regenerated every time the **LIST PSW[E] rborssrbaddress** command is issued wherein an **rborssrbaddress** operand is explicitly given. (It is not regenerated when the LIST PSW[E] command is given without operands.)

This equate labels the address of the next instruction then will be executed when the RB or SRB that owns the listed PSW resumes execution.

RB#n - This is a series of equates giving the locations and assumed lengths of each Request Block queued from a selected TCB in the Target Address Space. These equates are recreated each time the **LIST RBS** command is used. **n** corresponds to the sequence numbers shown in the **LIST RBS** display.

#REGS - This equate gives the location and length of the system control block field in which the:

LIST REGS
LIST RHREGS and
LIST RWREGS

commands found the low halves of general register data that they displayed. This equate is regenerated each time any of the above three commands is issued with an **rbaddress** or **ssrbaddress** operand. It is **not** regenerated when the command is given **without** operands.

#RHREGS - This equate gives the location and length of the system control block field in which the:

LIST REGS
LIST RHREGS and
LIST RWREGS

commands found the high halves of general register data that they displayed. This equate is regenerated each time any of the above three

commands is issued with an **rbaddress** or **ssrbaddress** operand. It is **not** regenerated when the command is given **without** operands.

#Rn - This is a series of equates that label the sixteen locations pointed to by the low half views of the sixteen general registers. These equates are (re)generated each time any of the:

LIST REGS
LIST RHREGS and
LIST RWREGS

commands are issued with an **rbaddress** or **ssrbaddress** operand. They are **not** regenerated when the command is given **without** operands.

#RHn - This is a series of equates that label the sixteen locations pointed to by the high half views of the sixteen general registers. These equates are (re)generated each time any of the:

LIST REGS
LIST RHREGS and
LIST RWREGS

commands are issued with an **rbaddress** or **ssrbaddress** operand. They are **not** regenerated when the command is given **without** operands.

#RWn - This is a series of equates that label the sixteen locations pointed to by the entireties of the sixteen general registers. These equates are (re)generated each time any of the:

LIST REGS
LIST RHREGS and
LIST RWREGS

commands are issued with an **rbaddress** or **ssrbaddress** operand. They are **not** regenerated when the command is given **without** operands.

SCB#n - This is a series of equates giving the locations and lengths of each STAE Control Block (SCB) queued from a selected TCB in an accessible address space. These equates are recreated each time the **LIST ESTAES** command is used to display a task's SCB queue. **n** corresponds to the sequence numbers shown in the **LIST ESTAES** display.

SSCT#n - This is a series of equates giving the locations and lengths of each Subsystem Control Table (SSCT) currently registered in the system. These equates are sequenced in queue position order. They are rebuilt every time the **LIST SSCT** command is issued.

SSRB#n - This is a series of equates giving the locations and lengths of each SSRB found in a selected address space. These equates are recreated each time the **LIST SSRBS** command is used. **n** corresponds to the sequence numbers shown in the **LIST SSRBS** display.

SSRBEP#n - This is a series of equates giving the locations of the entry points for each suspended SRB routine found in a selected address space. These equates are recreated each time the **LIST SSRBS** command is used. **n** corresponds to the sequence numbers shown in the **LIST SSRBS** display.

SSRX#n - In z/OS R1.13 and newer systems, this is a series of equates giving the locations and lengths of each SSRX found in a selected address space. These equates are recreated each time the **LIST SSRBS** command is used. **n** corresponds to the sequence numbers shown in the **LIST**

SSRBS display.

SSVT#n - This is a series of equates giving the locations and lengths of each Subsystem Vector Table (SSVT) currently registered in the system. (Most SSCTs contains pointers to SSVTs, some do not.) These equates are sequenced in SSCT queue position order. They are rebuilt every time the **LIST SSCT** command is issued.

STCB#n - This is a series of equates giving the locations and lengths of each Secondary Task Control Block in the Target Address Space. These equates are recreated each time the **LIST TASKS** command is used. **n** corresponds to sequence numbers shown in the **LIST TASKS** display.

SUSE#n - This is a series of equates giving the locations of each user data area that the registered SSCTs might point to. (Many SSCTs contains pointers to SUSEs, some do not.) These equates are sequenced in SSCT queue position order. They are rebuilt every time the **LIST SSCT** command is issued.

TCB#n - This is a series of equates giving the locations and assumed lengths of each Task Control Block in the Target Address Space. These equates are recreated each time the **LIST TASKS** command is used. **n** corresponds to sequence numbers shown in the **LIST TASKS** display.

VS#n_GC.varname [Global Constants Pool]
VS#n_SV.varname [Subroutine Variables Pool]
VS#n_LV.varname [Local Variables Pool]
VS#n_LV.area#.varname [Block Variables Pool]

These equates are applicable to debugging LE compliant programs. They are created by the **LIST VSTACK** command. For each Stack Frame level in which variable pools exist these equates label the starts and lengths of those pools.

Help EQuates Floating

The **EQUATE** command can be used to assigned either a "fixed" or "floating" base to the label being created:

- When a **fixed base** is assigned, the address expression that you give for the equate is evaluated immediately and the result is assigned as the equate's address. The equate will then continue to represent that particular location until it is deleted.

- When a **floating address** is assigned, the address expression that you give for the equate is **not** immediately evaluated. Instead, it is saved as part of the equate's definition, and it is automatically reevaluated each and every time the equate might be needed. Specifically:

  - The base address expression is evaluated any time the equate it used in address expressions.

  - It is also evaluated every time storage is displayed (via the **FORMAT,**

WHERE, and other commands) just in case the equate might currently represent a location being displayed.

The defining address expression for a floating equate may be **any** valid address expression! Examples:

- It might reference a **register.** Example: **EQUATE MYQNTRY R9% F** assigns an equate named **MYQNTRY** to represent whichever queue entry is currently pointed to by general register R9, **regardless** of how often the pointer in R9 is changed by the user program.

- It might reference a changing pointer in storage. Example **EQUATE CURRENT_TCB 21C% F** creates a label to represent the current task control block regardless of under which task Z/XDC gains control (in a multi-tasking program).

- It might reference either another **floating equate** or a field located within a **floating map.** For Example, **EQUATE NEWEST_RB .TCBRBP% F** creates an equate to always represent the newest RB queued from whichever TCB is currently represented by the map that contains the **TCBRBP** field. This will be true even if the TCB map is moved from one TCB to another and even if the queue of RBs changes from moment to moment.

The address expression that defines a floating equate may, itself, reference floating maps and other floating equates. By this means, an arbitrarily complex structure of floating maps and equates can be created, all being ultimately dependent upon just one anchoring pointer. Then if that single pointer should happen to change, the **entire** structure will automagically move in reaction to that single change.

An Example:

The following commands generate floating equates to represent the following:
- The first 10 Data Extent Blocks (DEBs) anchored from a TCB.
- The OPEN'd Data Control Blocks (DCBs) pointed to by each of those DEBs.
- The TIOT DD entries showing the DDNAMEs that have been OPEN'd by each of those DCBs.

EQ DEB TCB.TCBDEB% CFO=5 CFW=3 MAX=10 FLOAT HEX 50
This uses **autocloning** to generate 10 floating equates to represent up to ten OPEN'd **Data Extent Blocks** chained from an as yet unspecified TCB. Note, since these equates are floating, it is not actually necessary that the TCB map from which they are anchored be present as yet.

```
EQ DCB01 DEB01+18% FLOAT HEX 60
EQ DCB02 DEB02+18% FLOAT HEX 60
EQ DCB03 DEB03+18% FLOAT HEX 60
EQ DCB04 DEB04+18% FLOAT HEX 60
EQ DCB05 DEB05+18% FLOAT HEX 60
EQ DCB06 DEB06+18% FLOAT HEX 60
EQ DCB07 DEB07+18% FLOAT HEX 60
EQ DCB08 DEB08+18% FLOAT HEX 60
```

EQ DCB09 DEB09+18% FLOAT HEX 60
EQ DCB10 DEB10+18% FLOAT HEX 60
These generate 10 floating equates, each representing a **DCB** pointed to by a corresponding DEB.

EQ DD01 TCB.TCBTIO%+X2(DCB01+28) FLOAT HEX [follow shortcut for X2(...) help]
EQ DD02 TCB.TCBTIO%+X2(DCB02+28) FLOAT HEX
EQ DD03 TCB.TCBTIO%+X2(DCB03+28) FLOAT HEX
EQ DD04 TCB.TCBTIO%+X2(DCB04+28) FLOAT HEX
EQ DD05 TCB.TCBTIO%+X2(DCB05+28) FLOAT HEX
EQ DD06 TCB.TCBTIO%+X2(DCB06+28) FLOAT HEX
EQ DD07 TCB.TCBTIO%+X2(DCB07+28) FLOAT HEX
EQ DD08 TCB.TCBTIO%+X2(DCB08+28) FLOAT HEX
EQ DD09 TCB.TCBTIO%+X2(DCB09+28) FLOAT HEX
EQ DD10 TCB.TCBTIO%+X2(DCB10+28) FLOAT HEX
These generate 10 more floating equates, each representing a **TIOT DD entry** that has been OPEN'd by a corresponding DCB.

At this point, a floating structure of equates has been created, but none of the equates yet represents actual locations in storage. A **LIST EQUATES** command at this point would report DBC943W BASE RESOLUTION ERROR for each of these equates, and any attempt to use them in an address expression **(FORMAT DD10,** for example) would fail. The following commands correct this condition.

DMAP XDCMAPS.TCB [follow link for info about nicknames]
USING TCB 21C? FLOAT
These commands load a dsect map for the TCB and assign it to represent the current task.
- At this point, as many **DEBnn** equates (up to 10) will resolve successfully, as there are DEBs on the task's DEB chain.
- And that will lead to the successful resolution to the corresponding **DCBnn** and **DDnn** equates.
- At this point a **FORMAT DD01** command would be successful.
- And the resolved labels would automatically appear in displays of the locations to which they resolve.

And Furthermore...

LIST TASKS JES2
Security and authority permitting, this command produces a display showing the organization of Task Control Blocks (TCBs) in JES2's address space. It also automatically generates a series of equates, named **TCB#n,** to represent the addresses of those TCBs.

USING TCB TCB#3
Security and authority permitting, this command assigns the TCB map to represent JES2's jobstep TCB.

Doing this then causes the entire structure of floating equates (created earlier in this example) to immediately be reassigned to represent the DEBs, DCBs, and TIOT DD entries for the first 10 OPEN'd datasets in JES2's address space.

Help EQuates Autocloning

Autocloning is a method by which you can automatically generate an entire series of equates or dsects maps to represent either each entry in a table or each control block on a chain. The equates or dsects that are created all have unique sequence numbers but share a common root name that you must provide.

Chains

The **EQUATE** and **USING** commands have operands that let you define the following about chains:

- Where the chain field is.
- How wide the chain field is (3, 4, or 8 bytes).
- A mask for selecting a chain field's significant bits.
- The chain field value that signifies end-of-chain (zero, usually).
- The maximum number of chain entries to label or map.

Tables

z/XDC's autocloning supports tables of contiguous entries containing either fixed length or variable length entries. When variable, the table entries will (presumedly) have a length field. This length field can be of any reasonable width, and it may define either the table entry's entire length or only the length of the entry's variable portion (in which case, all entries in the table are presumed to consist partly of common fields of a fixed size). In support of all this, the **EQUATE** and **USING** commands have operands that let you define the following about tables:

- For fixed length table entries, or for variable table entries containing a fixed length root section, you can specify the length either of the entries or of the root section.
- For variable length table entries, you can specify where the length field is and how wide it is (1 through 8 bytes).
- You can specify either the length field value (if any) that identifies the end of the table or the address at which the table ends.
- Finally, you can specify the maximum number of table entries to label or map.

Whenever a series of equates or maps is created via autocloning, Z/XDC first checks to see whether or not a series of previously created equates or maps, having the same root name, already exists. If so, then that series is purged in its entirety prior to the creation of the current series.

The equates or dsects that are generated through autocloning can all have the same attributes and characteristics that any other equate/dsect might have. In particular, they can be assigned either fixed bases or **floating bases,** thus the equates/dsects can be created to represent either particular instances or generic instances of table or chain entries.

Floating maps and equates are a very powerful and useful capability, but you need to be aware that the resolution of floating bases is extremely **CPU intensive** and has to be done repeatedly during typical Z/XDC processing. Therefore, if the number of

equates and maps having floating bases grows "too large", then **Z/XDC's performance can be impacted severely!**

Unfortunately, the autocloning process has the potential of creating very large numbers of equates and maps. If it is used to create a **large number** of floating maps or equates, then Z/XDC's responsiveness may become unacceptably sluggish. So the **FLOAT** operand should be used with great care! Typically, you probably should never create more than a few dozen floating equates and maps, in total.

Help EQuates Ritargets

This topic has not yet been written.

Help EXEcutionlevels

The Elements of an Execution Environment

A program's execution environment consists of many elements. Major ones are:
- The next instruction's address.
- The condition code.
- The execution key.
- The key-mask.
- The Authorization Index (AX)
- The Extended Authorization Index (EAX)
- The addressing mode (31-bit or 24-bit or 64-bit).
- The Address Space Control Mode (Primary, Secondary, AR, or Home).
- The program mask.
- The DAT mode (on or off).
- The general, access, floating point and control registers.
- The linkage stack.
- The cross-memory access mode.
- The Primary Address Space.
- The Secondary Address Space.
- The Home Address Space.
- The Request Block queue.
- Locks held

Error Level vs. Retry Level Environments

When a program ABENDs (or reaches an **0C1-type** breakpoint), information about the execution environment at the time of the ABEND is saved by z/OS and can be displayed by Z/XDC. This is referred to in Z/XDC documentation as the **error level** execution environment.

You can tell Z/XDC (via the TRACE or GO/GOT/GOX command) to let the ABENDed program

resume execution; however, it may **or may not** be possible (and it may or may not be **desirable)** for the user program to resume in the same execution environment in which it ABENDed. This depends upon decisions that are made by the System's Recovery/Termination Manager (RTM) and that are beyond Z/XDC's direct control.

In any case, the execution environment that **can** be resumed is called the **retry level** environment.

Whenever an ABEND occurs and Z/XDC receives control, it is **very important** for the user to be aware of:

- What the error level environment is.
- What the retry level environment is.
- Whether the two environments are the same or **different.**

In particular, if you're using **0C1-type** breakpoints and you place a breakpoint into code for which Z/XDC has not been properly set up as its **newest** Recovery Routine, you will not be able to resume your program's execution after the breakpoint occurs! (Note, this restriction does not apply for TRAP2-type breakpoints... Read on.)

Error and Retry Levels for TRAP2-Type Breakpoints

Error and Retry level separation **never** occurs for TRAP2-Type breakpoints!

It is important to understand that this distinction between an error level and retry level environments is an **artifact** created by the System's **Recovery/Termination Manager** (RTM). If Z/XDC has received control via a mechanism that does **not** involve an ABEND or program check, then the error level vs. retry level distinction **collapses completely!** There is no distinction. They are one and the same.

This is what happens when **TRAP2-type** breakpoints are used. Z/XDC receives control as a **Trap Handler,** not as an ABEND Recover Routine. So no distinction between an error level and a retry level exists. So you will **always** be able to resume the program into which you placed the breakpoint.

This does **not** mean that using TRAP2-type breakpoints eliminates the need to understand the ins and outs of Execution Environments. This is because even when you are using TRAP2-type breakpoints, you **still** need to set up Z/XDC as your program's newest recovery routine because... What if your program suffers an ABEND or program check that is not a breakpoint? Who's going to handle that if Z/XDC is not your program's ESTAE?

But that's not the only reason... For complete details, see **HELP BREAKPOINTS TYPES.**

Error and Retry Levels During Dump Analysis

If you have licensed Z/XDC's **dump/XDC** feature, then you will be able to use Z/XDC for analyzing IPCS storage dumps. (See **HELP DUMPS** for more information.)

Dumps, of course, are just snapshots in time. Although execution threads might be discernible, those executions cannot be resume. So when it comes to ABENDed threads,

their error levels are quite discernible, but their retry levels are not.

But the need for a defined retry level is rather deeply baked into Z/XDC, so for dumps, dump/XDC considers the retry level to be the same as the error level, and that is what it tells Z/XDC. For more information, see **HELP DUMPS EXECUTIONTHREAD ERRORANDRETRYLEVELS.**

More About Error and Retry Levels

RTM uses the error vs. retry level distinction to control what execution environments an ABEND Recovery Routine (an ESTAE, for example) is and is not allowed to resume the execution of.

Generally (there are some exceptions), a Recovery Routine is allowed to resume only the code that **owns** that Routine. For ESTAEs and ESTAEXs, the owner will be the code that created the Recovery Routine in the first place (i.e. that issued the ESTAE[X] macro). For ESTAIs, the ownership issue is a bit more complicated.

Regardless, the resumable environment is what is called the **retry level environment.**

Why the hassle? Well, when a problem state program (for example) calls a supervisor state System Service and that service ABENDs, it would be an **integrity exposure** if the RTM allowed a Recovery Routine created by a problem state program to resume the execution of a supervisor state System Service.

Such integrity exposures are avoided by the RTM's careful attention to what is and is not a resumable execution environment for any given Recovery Routine.

The following commands are particularly useful for identifying and understanding the current error level and retry level execution environments.

LIST MSGS
LIST RBS
LIST LSTACK

In particular, whenever Z/XDC receives control for reasons that are a "surprise" to you, **LIST RBS** should be one of the very first commands that you issue! (See below for more information.)

Elements of an Execution Environment

Both the error level and retry level execution environments have their own PSWs, general registers, access registers, and all the other characteristics listed above. Accordingly, Z/XDC provides ways by which these various objects can be referenced and displayed.

Commands Pertaining to Error Level Objects

Generally, **error** level objects have names in Z/XDC that start with the letter **E.** Specifically, the following commands can be used to display error level objects:

LIST EPSW/EPSWE - Displays the error level Program Status Word.

FORMAT EPSW?/EPSWE?	- Displays the next instruction (in the error level Primary Address Space) that would have been executed if an ABEND or breakpoint had not occurred.
EWHERE	- Also displays the next instruction (in the error level Primary Address Space) that would have been executed if an ABEND or breakpoint had not occurred.
LIST EREGS/ERHREGS/ERWREGS	- Displays the error level general registers.
LIST EAREGS	- Displays the error level access registers.
LIST ECREGS/ECRHREGS/ECRWREGS	- Displays the error level control registers.
LIST FREGS	- There is no error vs. retry level distinction for floating point registers.
LIST ER0/EHR0/EWR0 etc.	- Displays error level general register R0, etc.
LIST EAR0 etc.	- Displays error level access register AR0, etc.
LIST ECR0/ECHR0/ECWR0 etc.	- Displays error level control register CR0, etc.
LIST FR0 etc.	- Displays error level floating point register FR0, etc.
LIST XMS	- Shows information about both the error level and retry level cross memory environments.

Commands Pertaining to Retry Level Objects

The following commands can be used to display retry level objects.

LIST PSW/PSWE	- Displays the retry level Program Status Word.
FORMAT PSW?/PSWE?	- Displays the next instruction (in the retry level Primary Address Space) that will be executed if user program execution is resumed (via Z/XDC's TRACE or GO command).
WHERE	- Also displays the next instruction that will be executed if user program execution is resumed.
FORMAT ALET(0)+...	- Displays locations within the retry level Primary Address Space.
FORMAT ALET(1)+...	- Displays locations within the retry level Secondary Address Space.
FORMAT ALET(2)+...	- Displays locations within the Home Address Space. (The Home Address Space is always the same space for both the error level and retry level execution environments.)
LIST REGS/RHREGS/RWREGS	- Displays the retry level general registers.
LIST AREGS	- Displays the retry level access registers.
LIST CREGS/CRHREGS/CRWREGS	- Displays the retry level control registers.
LIST FREGS	- There is no error vs. retry level distinction for floating point registers.
LIST R0/RH0/RW0 etc.	- Displays retry level general register R0, etc.
LIST AR0 etc.	- Displays retry level access register AR0, etc.
LIST CR0/CHR0/CWR0 etc.	- Displays retry level control register CR0, etc.
LIST FR0 etc.	- Displays retry level floating point register FR0, etc.
LIST XMS	- Shows information about both the error level and retry level cross memory environments.

When the Error and Retry Levels are the Same

When the error level and retry level environments are the same, the displays of error level objects generally will show the **same** information as displays of retry level objects. These displays will **differ,** however, in the following respects:

- If Z/XDC's **ZAP** command is used to change the contents of a retry level PSW, general register or access register, the corresponding error level object is not affected.

- When a program check type ABEND occurs (0Cx ABEND), Z/XDC automatically adjusts the retry level PSW to point to the **start** of the machine instruction that failed. (Z/XDC leaves the error level PSW unchanged usually pointing past the failing instruction.)

- When a **DEAD trap** is reached, Z/XDC automatically adjusts the retry level PSW to point past the DEAD trap.

- When execution reaches a Z/XDC **breakpoint,** Z/XDC automatically adjusts the retry level PSW to point to the start of the machine instruction at which the breakpoint was located. (The EPSW will be left pointing 2 bytes [regardless of instruction length] past the start of that instruction.)

Note that the above described automatic PSW adjustments are performed **only** when the error and retry level execution environments are the same. They are not done when the environments are different.

When the Error and Retry Levels are Different

The error level execution environment can be resumed only when it is **the same** as the retry level environment. So when it is possible to resume the error level environment, Z/XDC notifies you by automatically issuing a message saying:

DBC831I THE ERROR LEVEL AND RETRY LEVEL ENVIRONMENTS ARE THE SAME

On the other hand, if the error level execution environment **cannot** be resumed, then Z/XDC automatically issues a different message:

DBC954W THE ERROR LEVEL AND RETRY LEVEL ENVIRONMENTS ARE *DIFFERENT*

DBC831W detailed commentary

The **LIST MSGS** command can be used to redisplay this and other status messages at any time.

About Request Blocks and the Execution Environment

In most cases the error level and retry level execution environments are represented by specific points in the current task's **Request Block** (RB) queue. This queue can be displayed by the **LIST RBS** command:

- The RB at the retry level point in the queue will be captioned **(RETRY LEVEL).**

- If **(and *only* if)** the error level is **different** from the retry level, then the error level point in the RB queue will be separately captioned as **(ERROR LEVEL).**

When Z/XDC is entered as an ABEND Recovery Routine, the error level Request Block will **always** be the 3rd to last RB displayed by the LIST RBS command. The two newer RBs shown **did not exist** at the time of the ABEND! They were created by z/OS **after** the ABEND occurred in order to process the ABEND and to run Z/XDC. Consequently, Z/XDC captions it with **(ARTIFACT).**

Notice! Z/XDC is entered as an ABEND Recovery Routine when an ABEND has occurred, and when an 0C1-type breakpoint has been reached by execution. **However,** when execution reaches a **TRAP2-type** breakpoint, Z/XDC is entered as a **Trap Handler,** not as an ABEND Recovery Routine. In this case:

- There is no Error Level RB separate from the Retry Level RB.
- The Error Level and Retry Levels are always one and the same.
- There are no **artifact** RBs.
- The newest RB shown will always be the retry Level RB.

Resume Address is Hilighted in FORMAT Command Displays

The location pointed to by the retry level PSW is the user program's **resume address.** Whenever the **FORMAT** or **WHERE** command is used to display storage that includes the resume address, the machine instruction at the resume address is shown **hilighted.** Also, a **>** character is inserted into the display to further accentuate that location.

If the resume address, for some reason, falls into the **middle** of a machine instruction (or data area), then a **}** character is displayed instead of the >.

If you ask Z/XDC to resume user program execution (using a **TRACE** or **GO** command), the following things happen:

- The System will purge the **Request Blocks queue** of all RBs shown below (i.e. are **newer** than) the retry level RB.

- Then a Z/XDC service routine will be invoked to restore the **Linkage Stack,** and the **registers** (all types) and all other state information to values and states that are correct for the retry level environment.

- Finally, control will be returned to the user program so that it may resume execution.

Notice! when a Request Block is **purged** by the System, this means that the routine that it represents is **instantly** aborted. It is not allowed to execute any more instructions, and its Request Block is removed from the RB queue and destroyed.

When the retry and error level execution environments are the same, you have full flexibility either to trace user program execution (via the **TRACE** command) or to resume user program execution (via the **GO** command) as you see fit. **However,** when the two environments **differ,** it is not possible to trace or resume the error level environment! The error level environment (including the error level RB) **is purged!** Only the retry level environment can be resumed.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

RBS - What Request Blocks are, why z/OS use them, and why understanding RBs is essential for effective debugging.
ERRORLVL - Specific description of the error level environment.
RETRYLVL - Specific discussion of the retry level environment and why it can differ from the error level environment and what to do about it.

Help EXEcutionlevels RBs

Request Blocks are system control blocks that are very important to the way in which z/OS controls and keeps separate the execution of user programs, SVC routines, other system service routines, and user exit routines. They also are important in the management ABENDs, the assignment of blame, and the containment of their damaging effects.

When a user program fails with an ABEND, the ABEND may occur either in the program's code or in one of the program's exit routines, or in a system service routine. Failure in a system service routine may be due either to a bug in the service routine or (more likely) to invalid data being passed to the service routine from the user program.

For a full understanding of the nature of and reasons for an ABEND, it is vitally important to understand what Request Blocks are, how they are used by z/OS, and what they mean when they exist. Quite simply, the reason for this is that Request Blocks provide a specific and highly informative record of:

- Exactly what routines are executing at the time of an ABEND,
- The order in which they were invoked,
- Why they were interrupted,
- What their registers contain,
- What their PSWs are, and
- What instructions will be executed next if and when the routines are able to resume execution.

Request Blocks are **not** used for keeping track of programs that call each other via branching instructions (BR, BALR, etc.). Instead, they are created in z/OS on behalf of user programs when the programs use the ATTACH, LINK, and SYNCH SVC instructions. They also are created by the System for running many (but not all) SVC service routines. They also are created for running various user exit routines and for running the System's ABEND management routines (known as the "Recovery/Termination Manager" or "RTM" for short).

Request Blocks ("RBs") and Task Control Blocks ("TCBs") are the system control blocks that z/OS uses to control the execution of user programs and the various exit routines and system service routines that they invoke.

- The TCB is a "dispatchable" control block. It is a parameter block used by the

System’s dispatcher routines to define an independent (or semi-independent) unit of work in z/OS. In other words, it defines an "execution thread" for the Operating System. Accordingly, if two different programs (or two parts of the same program) are running under two different TCBs, then they can run simultaneously. The execution of each can proceed without regard to the execution of the other. It is possible for any number of TCBs to be running within any job or any TSO session.

- The RB specifically identifies an executable program or other unit of work running under a TCB. The RB contains a save area for the program’s PSW and registers. It controls whether or not the program is running in supervisor state. It allows non-authorized programs to call authorized system services (such as OPEN, CLOSE, and other SVC routines). It also allows authorized system services to call non-authorized user exit routines such that authorization is restored when the exit routine finishes. In other words, Request Blocks keep the authority levels of System routines and user routines separate such that one can call the other without conveying its authority on the other, and the other can return to the one without letting its authority leak back to the caller!

  - **Example:** The System’s OPEN SVC routines, which run authorized, can call a user defined DCB exit routine, which runs non-authorized. When the exit routine finishes, OPEN can resume authorized execution. This is made possible because OPEN, which runs under the control of a privileged Request Block, causes the DCB exit routine to run under the control of a nonprivileged Request Block. When the exit routine finishes, its nonprivileged Request Block is deleted, allowing the OPEN service routines to resume execution under its privileged RB.

RBs are queued to TCBs. z/OS looks to the TCB to determine that executable work exists. z/OS looks to the RB to determine the work’s current point of execution, its registers, its authorization state, and other characteristics.

When more than one RB is queued to a TCB, only the newest queued RB can execute. All older RBs are "interrupted" and cannot execute until the newer RBs are removed from the RB queue either by completing or by being purged.

RBs that are executing can be "interrupted" either voluntarily or involuntarily:

- When a program, running under the control of an RB, issues an SVC instruction, that program is interrupted, and the System builds and queues a newer RB for running the SVC routine requested by the SVC instruction. The program that issued the SVC cannot resume execution until the SVC routine completes execution and its RB is removed from the queue. This is a voluntary interruption.

  (Notice that the program might or might not be authorized, but the SVC routine being called will be authorized, and when the SVC routine completes, control will return to its issuing routine at whatever authority level it had when it issued the SVC instruction. This separation and preservation of authority level is made possible because it is the Request Block that keeps track of who has what authority.)

- Suppose that a program had issued a STIMER macro to create a "timer exit routine" that is scheduled to execute "later". Eventually, when "later" comes, a timer interrupt occurs, and the System "schedules" the timer exit routine for execution. This means that the System creates a type of RB called an IRB (an Interrupt Request Block) for controlling the execution of the timer exit routine. The System adds the IRB to the task’s RB queue thus making it the

newest RB on the queue. Being newest, the timer routine then executes. Meanwhile, the original program, being controlled by an RB that no longer is newest, is "interrupted" (i.e. frozen) at whatever point it happened to be executing, no telling where. It will remain frozen until the timer exit routine terminates and its IRB is then removed from the queue thereby allowing the program's RB to become newest again. This is a nonvoluntary interruption.

(Notice that if the STIMER macro were issued by a non-authorized user program, then the timer exit routine would also be non-authorized. It would execute non-authorized even if the Request Block that it interrupts happens to be running an authorized routine. Again, the Request Block separates and preserves authority levels!)

For a discussion of using Z/XDC to debug IRB routines, especially IRB routines that have interrupted an older Z/XDC being used to debug a normal routine, see HELP MULTITASK IRBS.

There are three different basic kinds of RBS:

PRB - **Program Request Blocks:** These are used to run user programs and certain kinds of exit routines. PRBs may or may not be privileged depending upon whether or not the user program is authorized. Examples:
- OPEN, CLOSE, and EOV ("DCB") exit routines.
- ABEND ("ESTAE", "ESTAEX", "ARR", and "ESTAI") exit routines.
- End of TASK ("ETXR") exit routines.

SVRB - **Supervisor Request Blocks:** These are used by z/OS to run certain SVC routines (but not all SVC routines). SVRBs always run privileged. SVRBs also are used by the System to run the Recovery Termination Manager (RTM). He's the guy that handles program checks and other ABENDs.

IRB - **Interrupt Request Blocks:** These are used to run exit routines that can receive control asynchronously to the execution of the user program. That is, these exit routines can interrupt the user program's execution at arbitrary and unpredictable times. This is because they are triggered by events that occur outside of the user program and beyond its direct control. Examples:
- TIMER ("STIMER") exit routines.
- ATTENTION ("STAX") exit routines.

Warning: There is another system control block called the SRB or "Service Request Block". Except for its unfortunate name, SRBs have nothing whatsoever to do with the other kinds of Request Blocks discussed here. SRBs are **never** queued from TCBs. They must not be confused with normal RBs.

Usually, only one RB is queued to a TCB. However, when SVCs are issued or exits occur, additional RBs will be queued. It is possible for any number of RBs to be queued from a single TCB.

When multiple RBs are queued to a TCB, only the routine running under the **newest** queued RB can actively execute! All older RBs are "interrupted". They cannot execute until the newest RB's routine finishes and returns control to the System (usually via branching on R14 or issuing the "RB termination" SVC, SVC 3). When this happens, that newest RB is removed from the TCB's queue and is deleted from the System. Thus, the next older RB becomes newest, and so its routine can then resume execution at the point at which it was interrupted.

In other words, although it is possible for many different routines to be "running" under a single task, only one of those routines (the newest one) can be **active.** All older routines are interrupted and cannot resume until all newer routines complete and terminate.

RBs are queued to TCBs. The "TCBRBP" field points to the newest queued RB. The "RBLINK" field of each Request Block points to the next older queued RB. The RBLINK field of the oldest queued RB points back to the TCB. Thus, the RB queue always is circular, a mildly annoying design arising from decisions made in the pre-historic days of ancient operating systems (not MVT, not even MFT, but probably PCP itself!).

The queue of Request Blocks running under any task can be displayed by Z/XDC's LIST RBS command.

Example:

When a typical user program is running, it runs authorized or non-authorized under the control of a PRB, and that PRB is the only RB queued from the TCB. Assume for this example that the user program is running non-authorized.

- Suppose that the program issues an OPEN SVC (SVC 19). This causes the System to create an SVRB and to add it to the TCB's RB queue. The SVRB then allows the OPEN service routines to run authorized.

- Suppose that the DCB being opened specifies a user defined "open exit" routine. When the OPEN service routines discover this, they then have to pass non-authorized control to the exit routine. Because the authorization level has to change, OPEN cannot just branch to the exit routine. Instead, it has to create a separate, nonprivileged RB (a PRB in this case) to control the exit's execution. OPEN does this by issuing a SYNCH SVC (SVC 12). SYNCH itself is one of the SVC routines that does **not** run under any RB. Instead, it builds a nonprivileged PRB for running the exit routine.

- So at this point, the TCB has 3 RBs queued:
  - The oldest RB ("RB#1") is a PRB for running the user's program.
  - The next newer RB ("RB#2") is an SVRB for running the OPEN SVC service routines.
  - The newest RB ("RB#3") is a PRB for running the DCB exit routine.

  Of the 3 RBs, the **only** one that is actively running a program is the **newest** RB. The two older RBs are interrupted! Their routines cannot resume execution until all newer RBs finish and terminate.

- Eventually, the DCB exit routine finishes, so it terminates by returning via R14. This causes its PRB to be removed from the TCB's RB queue and deleted from the System. This then leaves RB#2 (OPEN's SVRB) as the newest RB remaining on the queue, so now the OPEN service routines are able to resume authorized execution.

- When OPEN finishes its processing, it then returns to a system routine that removes its SVRB from the queue. This leaves the user program's PRB (RB#1) as the only remaining (and, therefore, once again the newest) queued RB, so now the user program is finally able to resume non-authorized execution with the next instruction following the OPEN SVC (SVC 19).

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

RESUMEADDRESS - The process for resolving a Request Block's resume address

Help EXEcutionlevels RBs Resumeaddress

With z/OS R1.13, IBM introduced improved support for executing code located **above-the-bar** (i.e. in 64-bit storage). The implementation of this support, of course, required z/OS to deal with PSWs in their **wide (16-byte)** format, instead of the older 8-byte format. This inevitably required the creation of **new fields** for saving and restoring the wider PSWs.

In the case of Request Blocks, one of the new fields is located in the XSB (a logical extension of the RB), and it is named **XSBOPSW16.** It is the 16-byte wide analog of the older, 8-byte wide **RBOPSW** field.

So now there are **two** fields into which a PSW is saved: The 8-byte wide **RBOPSW** and the 16-byte wide **XSBOPSW16.** And that creates a problem for IBM...

Over the decades, numerous user programs and 3rd-party vendor products have found it useful to **directly modify** the RBOPSW field in order to alter a program's resume address. So now in order for such a program to continue to work correctly, it's going to have to modify **two** fields, not just one.

But of course there is no way that that many programs are going to be recoded in any orderly fashion. So programs will ABEND, and regardless of "fault", customers will be angered. And that's not good for anybody!

So IBM needed to come up with a way to detect when the RBOPSW field **changed unexpectedly** and then propagate that change into XSBOPSW16. And they did... As follows:

- When an RB is created:
  - An **8-byte wide** version of the interrupted program's PSW is stored (as always) in **RBOPSW.**
  - The corresponding **16-byte wide** version of the interrupted program's PSW is stored in **XSBOPSW16.**
  - An exact copy of the 8-byte RBOPSW is checkpointed in **XSB_ORIG_RBOPSW.**

- Eventually, the routine (SVC routine, ABEND processing, timer exit, ESTAE routine, DCB OPEN exit, whatever) to be run under the new RB is given control, and it does its thing.

- If the routine is authorized (such as a customer or product SVC routine would be), then one of its "things" might be to change the interrupted routine's resume address. Typically, it would do this by directly modifying the interrupted program's RBOPSW field to change its resume address. (Highly frowned

upon by IBM, but happens anyway. **IEARBUP** is an IBM sanctioned alternative. See below for more information.)

- When the routine finishes "doing its thing", it terminates back to the System, which then does the housekeeping necessary to discard the no longer needed new RB and to resume the interrupted program under the control of the older RB.

- If RBOPSW (in the older RB) had been changed, then the interrupted program might (depending upon the change) resume at a different address.

Prior to z/OS R1.13, this scheme worked just fine. However, now with the advent of the **XSBOPSW16** field, z/OS needs to do something more complicated than just load-and-go with RBOPSW. That "something more" is this:

- When an RB routine terminates back to the System, something called RB Exit Processing gets control.

- First, RB Exit Processing compares RBOPSW with the checkpointed copy in XSB_ORIG_RBOPSW. If they are **identical,** then **XSBOPSW16** is used, without further ado, to resume the interrupted program.

- However, if they differ, then **XSBOPSW16** is either **modified or discarded** (as follows) in favor of information from **RBOPSW:**

  - The **hi-order 33 bits** of **RBOPSW** (which includes the PSWAMODE31 bit, and the execution state, key, ASC-MODE, condition code, masks, etc.) are copied into the hi-order 33 bits of **XSBOPSW16.** (The PSWENOTZARCH bit is flipped, of course.)

  - The instruction address fields of **RBOPSW** and of **XSB_ORIG_RBOPSW** are compared:

    - If they differ by plus-or-minus **6 bytes or less,** then the difference is added to or subtracted from XSBOPSW16. (This adjustment is done even when the arithmetic causes the 2GIG-to-zero wrap boundary to be crossed, in either direction.) In other words, the delta is applied and above-the-bar resume addressing (if any) is preserved.

    - On the other hand, if they differ by **more** than plus-or-minus 6 bytes, then XSBOPSW16's PSWEIA is discarded, and RBOPSW's PSWIA is used in its stead. Note, if XSBOPSW16 had contained an above-the-bar resume address, that fact **is LOST!**

Bottom line: When Z/XDC needs to know the resume address for a given Request Block, it no longer is a simple process of just loading up RBOPSW and using that. Now there is a rather elaborate **resolution process** needed when RBOPSW and XSB_ORIG_RBOPSW differ.

Z/XDC knows what that process is, so it uses it whenever it needs to... Such has when processing:

- The various **LIST [E]PSW[E]** commands.
- The **LIST RBS** command
- The **GO/GOT/GOX** commands
- Etc.

In other words, what Z/XDC displays and uses is the **resolved** resume address, not

just simply the RBOPSW and/or XSBOPSW16 fields. For more information, see the Built-in Help topic for the command or process of interest.

IEARBUP

As noted above, IBM doesn't like it when programs modify **RBOPSW** directly, and they get downright nasty when you (well, I actually) consider extending that direct modification concept to the new **XSBOPSW16** and **XSB_ORIG_RBOPSW** fields! Especially since they have gone to the trouble of providing a service **(IEARBUP)** for us to use for this purpose.

IEARBUP performs the resume PSW modifications so that we don't have to. And unlike direct modification, IBM **promises** that it will always work. **IEARBUP** is documented in **z/OS MVS Programming: Authorized Assembler Services Reference, Volume 2 (EDT-IXG)** (SA22-7610). Good luck!

Help EXEcutionlevels Errorlvl

When a program ABENDs (or reaches a Z/XDC breakpoint), information about the execution environment at the time of the ABEND is saved by z/OS and can be displayed by Z/XDC. This is referred to in Z/XDC documentation as the "error level" execution environment.

z/OS, as much as it reasonably can, attempts to freeze all processing in the ABENDing task. It does this by creating a new SVRB (Supervisor Request Block) and adding it to the ABENDing TCB's RB queue. It then causes a system service routine known as the Recovery/Termination Manager ("RTM") to run under the control of the new SVRB. This has the following consequences:

- Execution of all older RBs, being older, is frozen. Their resume PSWs and register sets are available for examination.
- The newest of the older RBs (the one that is next older to the RTM's SVRB) is the RB that had been executing when the ABEND occurred. This then is the "error level" RB. Its registers and PSWs are referred to within Z/XDC as "EREGS" "EPSW", and "ER0", "ER1", etc.
- Regardless of the authorization level of the ABENDed RB, the RTM, running under its own SVRB, can run authorized.

The RTM then checks to see if any ABEND error recovery routines ("ESTAE", "ESTAEX", "ARR", and "ESTAI" routines) have been defined for the ABENDing task. If not, then the RTM takes a dump (if needed) and eventually terminates the task.

On the other hand, if recovery routines exist, then the RTM passes control to them one by one.

Each error recovery routine that receives control from RTM runs under a PRB that is next newer than the SVRB under which the RTM is running. RTM passes control to the recovery routine via a SYNCH SVC or a SYNCHX service routine. This causes the System to create a PRB for running the recovery routine. When the recovery routine finishes and returns via R14, it branches to an RB termination SVC (SVC 3) that deletes the PRB and allows the RTM to resume execution under its SVRB. RTM loops this way either until the recovery routines are exhausted or until one of them requests RTM to resume user program execution.

Since Z/XDC is designed to run as an ESTAE or ESTAI type error recovery routine, Z/XDC always receives control from the RTM pursuant to an ABEND (or a breakpoint which, as far as z/OS is concerned, is just an s0C1 ABEND). Z/XDC always runs under a PRB that is next newer than the RTM's SVRB and that is **two** newer than the RB that ABENDed. Consequently, whenever Z/XDC's **LIST RBS** command is used to display the Request Block Queue:

- Z/XDC will always be shown running under the newest (last displayed) RB.
- The RTM will always be shown running under the next older RB (an SVRB).
- The error level RB will always be next older to that.
- Since RTM's SVRB and Z/XDC's PRB did not exist until the ABEND (or breakpoint) occurred, and since at least those two RBs will be purged in the event you decide to issue Z/XDC's **GO** or **TRACE** command, those two RBs will be labeled with **(ARTIFACT)** to alert you to the fact that as far as your program is concerned, those two RBs did not and will not exist!

In other words, the error level RB will always be the 3rd to newest Request Block displayed by the LIST RBS command.

If the error and retry level execution environments are the **same,** then this third to newest RB will be captioned **(RETRY LEVEL)** [because I cannot put two captions in one place].

However, if the error and retry level environments are **different,** then this third to newest RB will be captioned **(ERROR LEVEL)** in the LIST RBS display, and the **(RETRY LEVEL)** caption will appear on an **older** RB.

Note, in order to use Z/XDC effectively, a user program's error recovery environment usually needs to be designed such that Z/XDC can be given control by RTM **first** ahead of all other ESTAEs and ESTAIs.

Z/XDC provides ways by which the various objects and characteristics that constitute the error level environment can be referenced and displayed. Generally, error level objects have names in Z/XDC that start with the letter **E.** Specifically, the following commands can be used to display error level objects:

LIST EPSW FORMAT - Displays the error level Program Status Word. The PSW contains:
- The next instruction address.
- The condition code.
- The execution key.
- The addressing mode.
- The Address Space Control Mode.
- The program mask.
- The DAT mode.

FORMAT EPSW? - Displays the next instruction (in the error level Primary Address Space) that would have been executed had the ABEND or breakpoint not happened.

EWHERE - Also displays the next instruction (in the error level Primary Address Space) that would have been executed had the ABEND or breakpoint not happened.

LIST EREGS - Displays the error level general registers.

LIST EAREGS - Displays the error level access registers.

LIST ECREGS - Displays the error level control registers.

LIST FREGS - Displays the floating point registers. (The f-p registers are always the same for both the error level and the retry level environments.)

LIST LSTACK ECR15? - Displays the current Linkage Stack Entry.

LIST XMS - Displays the current Cross Memory Environment for both the retry level and the error level environments.

LIST RBS - Displays the current Request Block queue. The error level RB will be the third to last RB displayed. That RB will be captioned either "(ERROR LEVEL)" or "(RETRY LEVEL)". If it is captioned "(RETRY LEVEL)", then the error level RB and the retry level RB are the same RB.

LIST LSTACK - Displays the current Linkage Stack. The error level stack entry will be the second to last displayed. [The last displayed stack entry will be captioned **(ARTIFACT)].** If only one stack entry is displayed, then the error level linkage stack is empty.

Help EXEcutionlevels REtrylvl

If you tell Z/XDC (via the TRACE or GO command) to let the ABENDed program resume execution, it may **or may not** be possible for the user program to resume in the same execution environment in which it ABENDed. In either case, the execution environment that can be resumed is called the "retry level" environment.

When the retry and error level execution environments are the same as each other, you have full flexibility either to trace or to resume user program execution as you see fit. However, when the two environments differ, then it is not possible to trace or resume the error level environment! Only the retry level environment can be resumed.

Whether the retry and error level execution environments are the same as or different from each other depends upon decisions that are made by the System's Recovery/Termination Manager (RTM) and that are beyond Z/XDC's direct control. (It generally is **not,** however, beyond the programmer's control. See HELP EXECUTIONLEVELS RETRYLVL ESTAES, or just keep reading.)

Sometimes, the retry level environment restrictions imposed by RTM are desirable, but sometimes they are not. The following is an example of when they are desirable.

- Suppose that the user program needs to open a dataset. Accordingly, it issues the OPEN SVC (SVC 19) to invoke the z/OS system routines that perform OPEN processing.

- Suppose, however, that even though the dataset remains cataloged on the system, nonetheless, it has been scratch from disk, so in fact it does not actually exist. (In other words, the system catalog lies.)

- When z/OS's OPEN processing routine discovers this, it issues the ABEND SVC (SVC 13) with ABEND code s213 in order to abort OPEN processing and, possibly, OPEN's caller as well: the user program.

- This causes the RTM to gain control and eventually to pass control to Z/XDC to debug the ABEND.

In this example, the error level execution environment is the OPEN SVC routine (the point at which it issued the ABEND SVC), its PSW, its registers, and its Request Block. **However,** the retry level environment is the user program, its PSW, its registers, and its Request Block! A **LIST RBS** command will clearly show this.

In this example, it is not possible **and** not desirable(!) for RTM to allow an ESTAE routine (such as Z/XDC) to recover and resume execution of the OPEN service routine! This is true for the following reasons:

- Typically, the user program and its ESTAE routine are non-authorized, while the OPEN service routines are authorized. It would be a security and integrity exposure for RTM to permit a non-authorized ESTAE routine to change registers and other data and then resume the execution of an authorized routine.

- In any case, in this example there is no bug in OPEN to be fixed! It is not OPEN's fault that the given dataset does not exist. Instead, it is the user program's "fault" that it instructed OPEN to open a nonexistent data set. Therefore, it is the user program that needs to be corrected, not the OPEN service routines.

In this example, the proper course of action might be to SWAP to ISPF, allocate the missing dataset, SWAP back to Z/XDC, and use Z/XDC's GO command to cause the user program to reissue the OPEN SVC instruction. This can be done because the retry level environment belongs to the user program.

The rules that the RTM follow to determine the retry level execution environment differ depending upon whether the recovery routine (such as Z/XDC) is running as an ESTAE type routine or an ESTAI type routine.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

TYPES - ABEND error recovery exit routine types.

ESTAES - Retry level determination for ESTAE type error recovery exit routines.

ESTAIS - Retry level determination for ESTAI type error recovery exit routines.

Help EXEcutionlevels REtrylvl Types

There are several specific kinds of ABEND error recovery exit routines that can be established by user programs, by exit routines, and by system service routines. Briefly, they are:

```
   ESTAE  - For use by "normal" programs, service routines, and exits.
   ESTAEX - For both "normal" programs, AR mode programs, and PC routines.
   ARR    - For PC routines.
   FESTAE - For SVC service routines.
   STAE   - Obsolete, should not be used.

   ESTAI  - For parent tasks attaching subtasks.
   STAI   - Obsolete, should not be used.
```

For the purposes of determining a retry level environment, these various routines fall into two categories:

- The first five above listed routines are "ESTAE type" (pronounced "e-stay").

- The last two listed routines are "ESTAI type" (pronounced "e-sty").

These various routines are created by the following mechanisms and have the following characteristics.

ESTAE
- Established by an ESTAE macro when it issues SVC 60.
- Is owned by the newest RB existing at the time that the ESTAE macro is executed.
- Also is owned by the newest linkage stack entry in existence at the time that the ESTAE macro is executed.
- Can be issued from any user program, system service routine, or exit routine running in Primary ASC-MODE and not running in any cross-memory mode.
- Provides protection in the current task on behalf of the program, system service routine, or exit routine from which the ESTAE macro was issued.
- Is an ESTAE type routine.

ESTAEX
- Established by an ESTAEX macro when it issues a PC instruction.
- Is owned by the newest RB existing at the time that the ESTAEX macro is executed.
- Also is owned by the newest linkage stack entry in existence at the time that the ESTAEX macro is executed.
- Can be issued from any user program, system service routine, or exit routine running in Primary ASC-MODE or Access Register ASC-MODE, and running in any cross-memory mode.
- Provides protection in the current task on behalf of the program, system service routine, or exit routine from which the ESTAEX macro was issued.
- Is an ESTAE type routine.

ARR
- Is associated with PC routines.
- Provides protection in the PC routine to which it is associated.
- Is owned by the PC-type linkage stack entry that was created for running the PC routine to which this ARR is associated.
- Is an ESTAE type routine.

FESTAE
- Can only be used by system service routines running under the control of an SVRB (such as an SVC routine or the RTM).
- Established by a FESTAE macro when it directly builds an SCB (STAE Control Block) within a special area of the current SVRB.
- Is owned by that SVRB.
- Is **not** owned by any linkage stack entry.
- Provides protection in the current task on behalf of the system service routine from which the FESTAE macro was issued.
- Is an ESTAE type routine.

STAE
- An archaic type that should no longer be used in current programming.
- Established by a STAE macro when it issues SVC 60.
- Is owned by the newest RB existing at the time that the STAE macro is executed.
- Also is owned by the newest linkage stack entry in existence at the time that the STAE macro is executed.
- Can be issued from any user program or exit routine running in 24-bit addressing mode, Primary ASC-MODE, and not running in any cross-memory mode.
- Provides protection in the current task on behalf of the program or exit

routine from which the STAE macro was issued.
- Is an ESTAE type routine.

ESTAI
- Established by an ATTACH or ATTACHX macro when an "ESTAI=" operand is given.
- Is owned by the parent task from which the ATTACH(X) macro was issued.
- Is **not** owned by any linkage stack entry.
- Provides protection in both the attached subtasks and **all** of that task's descendants on behalf of the parent task from which the ATTACH(X) macro was issued.
- Executes under whatever task is ABENDing.
- Is an ESTAI ("e-sty") type routine.

STAI
- An archaic type that should no longer be used in current programming.
- Established by an ATTACH macro when a "STAI=" operand is given.
- Is owned by the parent task from which the ATTACH macro was issued.
- Is **not** owned by any linkage stack entry.
- Provides protection in both the attached subtasks and **all** of that task's descendants on behalf of the parent task from which the ATTACH macro was issued.
- Executes under whatever task is ABENDing.
- Is an ESTAI type routine.

Z/XDC can be established as an ESTAE, ESTAEX, FESTAE, ESTAI or ARR routine. It cannot be used as a STAE or STAI routine.

Help EXEcutionlevels REtrylvl ESTAEs

The following are ESTAE ("e-stay") type recovery exit routines:
```
    ESTAE  - For use by "normal" programs services, and exits.
    ESTAEX - For both "normal" programs, AR mode programs, and PC routines.
    ARR    - For PC routines.
    FESTAE - For SVC service routines.
    STAE   - Obsolete, should not be used.
```
Z/XDC can be established as an ESTAE, ESTAEX, FESTAE, or ARR routine. It cannot be used as a STAE routine.

ESTAE type recovery routines provide ABEND protection in the **current** task both for the program or exit routine running under the current Request Block and for any future routines that may run under newer Request Blocks. (They do **not** provide ABEND protection for routines running under older Request Blocks (if any) and for routines running under other tasks.

z/OS considers ESTAE type routines to be "owned" by the current Request Block, i.e. by the newest Request Block in existence at the time that the ESTAE type macro is issued. Accordingly, RTM uses the rule that the retry level RB for an ESTAE type routine is the ESTAE's owning RB.

Because of this rule, **any** program, system service routine, or exit routine that runs under a Request Block can establish its own ABEND protection and can make itself be the retry level should an ABEND occur.

Accordingly, if you wish to use Z/XDC to debug any SVC routine that you might be

writing, or any exit routine that your program might use, or even many system exit routines, simply issue an ESTAE type macro making Z/XDC the ESTAE type recovery exit routine for the current code. By this method, Z/XDC can be used to debug any of the following:

- DCB exit routines.
- Attention (STAX) exits.
- Timer exits.
- End of task (ETXR) exit routines.
- Even other ESTAE and ESTAI type error recovery exit routines!
- PC routines.
- VTAM exit routines.
- Any user written SVC routine.
- Many IBM written SVC routines that you might wish to modify **provided** that it is not one of the several SVC routines that Z/XDC uses in its own processing: OPEN, CLOSE, DYNALLOC, TPUT/TGET, ESTAE, STAX, WTO(R), ENQ, DEQ, and others.
- SUBMIT command exit (TSO).
- SMF exit routines.
- DASDM exit routines (IGGPRE00 and IGGPOST0).
- Some subsystem exit routines.
- RACF exit routines.
- CICS exit routines.
- IMS exit routines.
- Third party vendor exit routines.
- And many others.

Briefly, Z/XDC can be established as an ESTAE routine using the following macro sequence:

```
   LOAD  EPLOC==CL8'XDC'   ("==" not a typo)
   ESTAE (R0)
   #DIE  'HIYA, HIYA'      (macro available from Z/XDC's XDCMACS library)
```

This example is an over simplification. For more complete information, please see HELP DEBUGGING EXITROUTINES.

(The XDCMACS library's factory default name is hlq.XDCV31.XDCMACS. Ask your Systems Programmer for its actual name at your Data Center.)

Please be careful not to confuse an ESTAE's "owning" RB with the RB under which the ESTAE might be executed. At the time that a program issues an ESTAE type macro in order to establish protection, no ABEND has yet occurred. Accordingly, the ESTAE type routine is not being executed, only "established", i.e. its availability is being made known to the System. The RB under which it might later be executed does not even exist yet.

Help EXEcutionlevels REtrylvl ESTAIs

The following are ESTAI ("e-sty") type recovery exit routines:

```
   ESTAI  - For parent tasks attaching subtasks.
   STAI   - Obsolete, should not be used.
```

Z/XDC can be established as an ESTAI routine but **not** as a STAI routine.

ESTAI type recovery routines are established when an ATTACH macro is issued with an "ESTAI=" (or "STAI=") operand specified. ESTAI type routines provide ABEND protection in **subtasks** of the current task. More specifically, they provide protection in both the subtask being attached and in **all** descendant tasks of that

task.

ESTAI type routines are "owned" by the attaching task; however, when a descendant task ABENDs, the ESTAI routine actually executes as part of the ABENDing task, **not** the owning task! In other words, the PRB under which an ESTAI routine executes is queued newest to the ABENDing TCB's RB queue.

Because ESTAIs are owned by TCBs and not by RBs, the simple rule that the RTM uses for determining an ESTAE's ("e-stay") retry level RB cannot be used for ESTAIs. Instead, retry level determination follows a more complex set of rules for ESTAIs:

- First, the RTM checks to see if the current ABEND occurred within an RB that is, or that is newer than, the PRB of another error recovery exit routine (i.e. if the current ABEND is a nested ABEND occurring in another ESTAE or ESTAI type routine that was already running to handle an earlier ABEND). If so, then the retry level RB will be that older error recovery exit's PRB, and all newer RBs will be deleted when program resumption is requested.

- Otherwise, the RTM scans the RB queue from oldest to newest looking for any RB that is **not** a PRB. When it finds one, then that RB and all newer RBs (including any additional PRBs there might happen to be) are deleted (when program resumption is requested), and the newest PRB not deleted is the retry level RB.

What do these rules mean?

- ESTAIs can be used to **examine** ABENDs occurring in routines running under **any** RB queued to any TCB in the protected branch of the subtask tree. **But** they can be used to **recover** from ABENDs occurring in routines running under only certain PRBs.
- In general, IBM has designed ESTAIs so that they can be used to recover from ABENDs occurring in user programs and their ESTAE and ESTAI type error recovery exit routines. ESTAIs **cannot** be used to recover from ABENDs occurring in any other kind of exit routine or occurring in any system service routine.
- Note that ESTAIs can be used to recover from ABENDs occurring only in certain kinds of routines running under PRBs. They cannot recover from ABENDs occurring in DCB exit routines, ETXR exit routines, etc. even though those routines also run under PRBs.
- ESTAIs cannot be used to recover from ABENDs occurring in any routine running under either an SVRB (SVCs) or an IRB (attention and timer exits).
- The oldest RB queued to any task is **always** a PRB. (This is because the ATTACH process not only creates a TCB for running a subtask, but also creates the TCB's first PRB for running the designated program in that subtask.) Accordingly, ESTAIs can be used to recover from ABENDs occurring directly in the attached program.
- If an attached program issues a LINK SVC (SVC 6) to call a second user program (a process which creates a second PRB), then an ESTAI routine can be used to recover from ABENDs occurring within the linked to program.
- Ditto if the 2nd program itself issues a LINK to a 3rd program, and so on.

When a debugging session is started via the XDCCALL utility program (or via any of its aliases: XDCCALLA, XDCCMD, and XDCCMDA), the utility attaches the user program as a subtask and specifies the "ESTAI=" operand to establish Z/XDC as an ESTAI routine for the user program. When Z/XDC is setup in this way, it can be used to debug the user program without necessarily requiring that the user program be modified, subject to the following considerations.

- If the user program does not issue any of its own ESTAE type macros, then Z/XDC can be used to debug the program's main code. It cannot, however, be used to debug any DCB exits, attention exits, timer exits, ETXR exits, or VTAM exits

unless those exits are modified to setup Z/XDC as their ESTAE routine.
- If the user program **does** issue ESTAE macros, then Z/XDC cannot be used to debug the program's main code (unless the user's ESTAE routine BALRs to Z/XDC or the ESTAE SVC instructions (SVC 60) are zapped to NOPs), **but** Z/XDC can be used to debug the ESTAE type exit routines themselves!
- If the user program's main code calls other programs via LOAD/BALR, LINK, or XCTL, Z/XDC can be used to debug the called programs without requiring any modifications.
- If the user program attaches subtasks, then Z/XDC can be used to debug the subtasks without requiring any modifications.

For more information about XDCCALL, see HELP XDCCALL.

Note, in those cases where Z/XDC cannot be immediately used to debug a piece of code, Z/XDC's HOOK command generally can be used to overcome the problem. See HELP HOOKS for more information.

When a debugging session is started via Z/XDC's Startup Panel in ISPF (using the panel's options 1 or 2, but not option 3), the panel internally uses the XDCCALL utility or one of its aliases to start the debugging session. Accordingly, the information discussed above applies to panel started sessions as well.

Help EXIts

Z/XDC recognizes and has interfaces to several exit routines (subroutines that Z/XDC will call under specific circumstances). Broadly speaking, exit routines fall into two categories: user writable and customer writable.

- A "user writable" exit is an exit that can be written by individual Z/XDC users. In other words, individual users can write their own private exit routines that Z/XDC will use only when Z/XDC is being executed by that user.

- A "customer writable" exit is one that can only be written by a customer's System Support staff and installed for use by all copies of Z/XDC regardless of who is executing Z/XDC and regardless of the name under which Z/XDC is executing.

Note that a user writable exit can always be made publicly available to all users.

Subtopics Index

For specific descriptions of the available exits and for other additional information, select the following topics. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip. Those topics flagged by **(*)** describe exits that require Z/XDC's Z/XDC's **Miscellaneous Exits Interface.** (See **MISCXITS** below.)

ACIF - Access Control Interface Exit: Called instead of the System's S.A.F. interface whenever Z/XDC needs to check a security rule. This is a customer writable exit. It cannot be written by individual users.

FRREND - End of FRR exit: Called when Z/XDC is running as an FRR and an END

command has been issued.

- **INIT** - **(*)** Initialization Exit: Called each time Z/XDC is called during a debugging session.
- **MISCXITS** - Miscellaneous Exits Interface: A common driver for certain other Z/XDC exits, specifically, those exits listed here and flagged by **(*).**
- **RESUME** - **(*)** Resumption Exit: Called each time during a debugging session that Z/XDC releases control back to the program being debugged.
- **TERMIN8** - **(*)** Termination Exit: Called if Z/XDC has decided to terminate the debugging session (e.g. because the user has issued an END command).
- **UCI** - User Communications Interface Exit: Issues commands to Z/XDC, receives displays back (i.e. replaces Z/XDC's normal terminal communications interfaces).
- **USERCMDS** - **(*)** User Commands Exit: Called whenever the user enters a command that Z/XDC does not recognized.
- **USVCTRCE** - **(*)** User SVC Trace Exit: Called when the user has issued a TRACE command and the current machine instruction is a user SVC (must be used if the SVC makes returns to locations other than the following machine instruction).
- **REGSTACK** - **(*)** User Register Stack Exit: Called by c/XDC when performing Metal C variable discovery to identify the general registers of the current Subroutine's caller. (Used with Metal C programs whose Prolog Logic has been replaced with code that uses **non-standard** register save areas.

Help EXIts Acif

A customer's System Support staff may, if desired, write an Access Control Interface Exit (an "ACIF" exit) for Z/XDC. Normally, when Z/XDC needs to check access to a system security rule, it calls the operating system's "S.A.F." interface. ("S.A.F." stands for "System Authorization Facility" and is always present in z/OS regardless of whether or not a security system [such as RACF, CA-ACF2, or CA-Top Secret] is installed). However, if an ACIF exit is provided, then Z/XDC will call that instead of S.A.F. The ACIF exit may itself call S.A.F, or it may simply make the security decision itself and return to Z/XDC without calling S.A.F.

A sample ACIF exit has been written showing the structural framework needed. The sample also contains commentary that should be helpful for writing your own version of the exit. It can be found in the member named SRCACIF of the XDCASM library. (The factory default name is hlq.XDCV31.XDCASM. Ask your Systems Programmer for the library's actual name at your Data Center.) To assemble the sample, you will need the macros provided in the XDCMACS library. (The factory default name is hlq.XDCV31.XDCMACS.)

The ACIF exit must be linkedited into its own load module, and that module must be named exactly XDCACIF regardless of whether or not Z/XDC itself has been renamed. XDCACIF must be a reentrant load module, and it must be located in common storage -

i.e. in the PLPA or on the system's Link Pack Queue (now known as the DLPA). Z/XDC will ignore any copy of XDCACIF that it finds in the private area.

Each time Z/XDC (or a clone) executes authorized, it checks for the presence in common storage of an XDCACIF load module and for whether or not the load module has been changed since the last time Z/XDC ran authorized. If found or if changed, then Z/XDC saves a pointer to XDCACIF in a protected CSA control block for use by all future security calls from **that particular clone** of Z/XDC.

(Note, a "clone" of Z/XDC is a Z/XDC that has been renamed to something other than "XDC", e.g. "X22", "XYZ", "U45", "ETC", etc.).

Accordingly, if a customer wishes to change its XDCACIF routine on the fly, it can do so just by loading a new copy of XDCACIF into common storage (via MODREP, OMEGAMON, etc.) and then running Z/XDC authorized. If multiple clones of Z/XDC are installed, then you will have to execute each clone authorized.

Note that issuing **xxxCALLA IEFBR14** from ISPF's Command Shell (=6) is a good way to run Z/XDC authorized. ("xxx" represents a Z/XDC clone's name).

INPUTS: When Z/XDC calls the ACIF exit, it provides the following inputs:

R0 - Points to a parameter list called the "XACPL" (mapped by the #DBCXACP macro). This parameter list contains information about:
- The kind of security system that Z/XDC has determined is being used by your system.
- The kind of security call being made.
- Various data that might be useful to the ACIF exit in making it's permit/deny decision.

The #DBCXACP macro can be found in the XDCMACS library. (The library's factory default name is hlq.XDCV31.XDCMACS. Ask your Systems Programmer for its actual name at your Data Center.) Note, the S.A.F. interface does not look in R0 for any input information. The XACPL that Z/XDC passes via R0 is strictly for the ACIF exit's benefit.

R1 - Points to the same RACROUTE parameter list that Z/XDC would have passed to S.A.F. if the ACIF exit were not present. Please note the following:
- All calls from Z/XDC are "REQUEST=AUTH" type calls which means that the RACROUTE plist always points to a RACHECK plist.
- The RACROUTE plist is documented by IBM's ICHSAFP macro found in SYS1.MACLIB.
- The RACHECK plist is documented by the ICHACHKL macro found in SYS1.MODGEN.
- The RACROUTE macro operands that Z/XDC uses as well as the data that they point to vary depending upon which security system Z/XDC determines is installed: RACF, CA-ACF2, CA-Top Secret, or none. Z/XDC's **Installation Guide** documents this information.

R13 - Points to a standard 72-byte register save area.
R14 - Points to a return address in Z/XDC.
R15 - Points to XDCACIF's entry address.

- Z/XDC may be either authorized or non-authorized when it calls XDCACIF.
- If authorized, then Z/XDC will be running in supervisor state and key 0.
- If non-authorized, then Z/XDC will be running in problem state and jobstep key (usually key 8).
- Other input states and modes:
  PASID=SASID=HASID
  Primary ASC-mode
  31-bit addressing mode

```
               Enabled, no locks held

OUTPUTS: When the ACIF exit returns to Z/XDC, it must restore all registers, states,
and modes except the following:
    R15 - Must be set as follows:
          0 - Access to the rule is permitted.
          4 - Access to the rule is not controlled. Z/XDC, therefore, will proceed
              as if the access were permitted.
          8 - Access to the rule is denied.

        - Return must be made to 0(,R14).
```

For more information, refer to the System Security sections of Z/XDC's **Installation Guide.** Also, refer to the SRCACIF member of Z/XDC's XDCASM library.

Help EXIts Frrend

Z/XDC calls this **End of FRR** exit whenever Z/XDC is running as an FRR and the user issues an **END** command.

Normally, when an **END** command is issued, the ABEND that Z/XDC was debugging is percolated. However, when Z/XDC is running as an **FRR,** percolation is not possible! Instead, the routine that had ABENDed now (one way or another) has to be unceremoniously terminated!

The reason why has to do with the fact that prior to giving control to Z/XDC proper, Z/XDC's FRR interface first had to exit the FRR environment in favor of a "retry" routine. (For information as to why, see HELP DEBUGGING FRR.) So by the time an **END** command is issued, the original ABEND environment no longer exists, so there is nothing that **can** be percolated!

Anyway, just prior to terminating the **SRB** or **RB** routine, Z/XDC checks to see whether or not an **FRREND** exit routine has been provided. If so, then it is called. This routine then gets to decide whether the termination that's about to happen is to be silent or big, noisy and messy. It does this by setting a return code into **R15:**

- **Silent:** (R15=0) A "normal" (i.e. nonABEND) termination will occur as follows:
  - For SRB mode code, a jump will be made to **CVTSRBRT** (z/OS's normal SRB termination handler).
  - For task mode code, an **SVC 3** instruction will be executed (z/OS's normal Request Block termination handler).

  Silent termination is the default, absent an **FRREND** exit routine.

- **Noisy:** (R15=4) A DEAD trap is executed to force an s0C1 ABEND. The DEAD trap contains a **DBC557T** message. See HELP MESSAGES DBC557 for more information.

In order to provide an **FRREND** exit, you must first provide a **DBCPARM** block. For FRRs, you do this by providing the 31-bit address of **DBCPARM** in the first word of the **FRRSPARM** area. See the **MVS Authorized Assembler Services Guide** ((SA22-7608) for information about the **FRRSPARM** area. See HELP MACROS #DBCPARM for information about the **DBCPARM** block.

In the DBCPARM block, there are two fields related to **FRREND** exit processing:

- **DPFRREND:** This field provides the address of your **FRREND** exit routine:
  - It must contain a positive value. If it contains a zero or negative value, it is ignored.
  - The address must be valid in the **home** address space.

- **DPFRREUD:** You may set this field to an arbitrary value to be passed into the **FRREND** routine.

Residency

The address in the **DPFRREND** field must be valid in the Home address space. Therefore, if the **DBCPARM** block that you provide resides in an aspace other than Home, then the **FRREND** routine must reside in common storage.

Input States and Registers:

FRREND is entered with the following settings:

AMODE: - 31-bit
ASC-Mode: - Primary
Cross memory: - PASID=SASID=HASID
Key: - Zero
Locks: - None held
Mode: - SRB mode (PSATOLD=0) or task mode (PSATOLD<>0)
State: - Supervisor

R0: - Zero. (This is intended to be a function code. So far the only defined code is zero, meaning that your routine is being called as an **FRREND** routine.)
R1: - Copy of the contents of **DPFRREUD**
R2: - Contains the address of a **copy** of the **SDWA** for the original ABEND that was being debugged.
R3: - Contains the user program's **primary ASN**
R4: - Contains the user program's **secondary ASN**
R13: - Address of a standard 72-byte register save area
R14: - Return address
R15: - FRREND's entry address

Output States and Data:

All state data must be returned unchanged. The registers must be set as follows:
R15: - This must contain one of the following return codes:
R15=0 - The user program is to be terminated silently.
R15=4 - The user program is to be ABENDed at a DEAD trap (containing message **DBC557T).**
R14 R0-R4 - Need not be restored.
R5-R13 - Must be restored.

Help EXIts Init

Under construction. Meanwhile, for more information, please:

- Refer to commentary located in the **#DBCPARM** macro located in the XDCMACS library.

  (The library's factory default name is **hlq.XDCV31.XDCMACS.** Ask your Systems Programmer for its actual name at your Data Center.)

- Contact ColeSoft. (See **HELP SUPPORT CONTACTUS** for how to do so.)

Help EXIts Miscxits

The **Miscellaneous Exits Interface** is an interface routine between Z/XDC and most of the user writable exit routines. Essentially, the Interface is a router. Whenever Z/XDC calls one of the supported user exits, it actually calls the router and passes to it a request code that identifies the specific exit to be called. The router must then either pass control forward to the desired exit or return to Z/XDC with a signal that the requested exit does not exist.

The user exits supported by the Miscellaneous Exits Interface are:

- Initialization
- Resumption
- Termination
- User SVC Trace
- User Commands (includes the rexx/XDC Interface)
- User Register Stack

All other Z/XDC exits use other interfaces.

Providing a Miscellaneous Exits Interface

A sample **Miscellaneous Exits Router,**
a sample **User Commands Exit,** and
a sample **User Register Stack Exit**
are available in the **SRCXITSR, SRCXUCMD** and **SRCXUREG** members (respectively) of the **XDCASM** library. (The factory default name is hlq.XDCV31.XDCASM. Ask your Systems Programmer for the library's actual name at your Data Center.)

Also, an **XDCXITS** load module, containing the sample router and sample exits, is available in the **XDCLINK** load library. (The factory default name is hlq.XDCV31.XDCLINK.)

The sample exits provided with Z/XDC do the following:

- The sample **User Commands Exit** implements two commands:

  - The **LIST TIOT** command displays information from any Task I/O Table queued from any TCB located in any accessible address space in the system.

  - The **UCTEST** command is an otherwise undocumented command for testing some of the call back interfaces to Z/XDC's support routines.

- The sample **User Register Stack Exit** implements knowledge of standard format-1 z/OS save area linkage. It must be used when debugging **Metal C** programs that use non-standard Stack Frames.

Z/XDC is distributed with these sample exits already installed and available for use.

You may write your own Miscellaneous Exits Interface routine, or you might take the sample routine and modify it. In either case, once you have built your own Miscellaneous Exits Interface Routine, you will have to let Z/XDC know that the routine exists and should be used. The way to do this (automatically or manually) varies depending upon how you have set up your Z/XDC debugging session:

- **Automatic:** If you start a Z/XDC debugging session by using the **XDCCALL** program (or any of its aliases) or by using the **Z/XDC Startup Panel** in ISPF, then Z/XDC is automatically established as an error recovery routine (an ESTAI) for your program. In this case you should linkedit your Miscellaneous Exits Interface routine into a load module named **xxxXITS** (where **xxx** matches Z/XDC's current name [usually **XDC**]) and place that load module into any library that is a part of your program's load module search order. That would include:
  - //TASKLIB
  - //ISPLLIB (if the debugging session is running under ISPF),
  - //STEPLIB
  - //JOBLIB
  - The PLPA libraries
  - Or the linklist libraries

  xxxXITS should be reentrant.

  If **XDCCALL** finds an **XDCXITS** load module, then it will load it into storage and set it up for use by Z/XDC. See **HELP XDCCALL** for more information.

- **Automatic:** If you start (or continue) a debugging session via a hook (using either the **HOOK** command or the **#XDCHOOK** macro), then the service routine that creates the debugging environment also looks for an **xxxXITS** module, and if it finds one, it LOADs it and sets it up for use by the debugging session. (For more information, see **HELP HOOKS.)**

- **Manual:** If you set up Z/XDC by issuing an ESTAE or ESTAEX macro or by using the ESTAI= operand of an ATTACH or ATTACHX macro, then in order to make a Miscellaneous Exits Interface routine available to Z/XDC, you must build a **DBCPARM** block and store a pointer to the interface routine into the block's **DBCXITSX** field. See **HELP EXITS #DBCPARM** for more information.

- **SET EXITS:** If it is impractical to utilize either of the other two exit installation techniques, the **SET EXITS ACTION=INSTALL** command is available. **SET EXITS** requests that Z/XDC make the necessary adjustments to the **DBCPARM** block

(or build one) so that a Miscellaneous Exits Interface routine is made available. **HELP COMMANDS SET EXITS** for more information.

Note that at Z/XDC installation time, the sample **XDCXITS** load module will automatically be made available for loading by the **XDCCALL[A]** program and, therefore, for use by Z/XDC. The sample XDCXITS contains code that implements a **LIST TIOT** command, so when XDCXITS is available, this command will be available too.

When Z/XDC is invoked **manually,** the **LIST TIOT** command will **not** be automatically available. To make it available, your program will have to use a LOAD macro to bring the XDCXITS load module into storage, and then store its pointer into the **DBCXITSX** field (of the **DBCPARM** control block) before issuing the ESTAE[X] or ATTACH w/ESTAI= macro for Z/XDC.

Coding Requirements for a Miscellaneous Exits Interface

If you do provide a Miscellaneous Exits Interface for Z/XDC's use, then Z/XDC calls that interface for **all** of the user exits listed above.

As stated above, your interface routine must determine the exit being called, then either pass control to code that performs the appropriate exit function, or return directly to Z/XDC with a return code that indicates that the exit does not exist, whereupon Z/XDC will perform its default action.

Because of a validity check requirement, Z/XDC ensures that the first machine instruction at the entry point for the Miscellaneous Exits module must be an R15-based unconditional branch, as follows:

```
        47F0 Fxxx     BC  15,xxx(,R15)
   or:  47F0 Fxxx     B   xxx(,R15)
```

If the Miscellaneous Exits module's first instruction is not as shown above, then the module will be ignored.

INPUTS: When Z/XDC calls the Miscellaneous Exits Interface, the registers and environment are set as follows:

asid - The Primary and Secondary Address Spaces are both set to being the Home Address Space. This is true even when the environment being debugged is a cross memory environment.

R0 - Contains a copy of the **DBCXITSD** field of the **DBCPARM** block. The Miscellaneous Exits Interface routine may change the contents of this field at any time for any purpose. Any such change will be preserved by Z/XDC. For more information, see: **HELP MACROS #DBCPARM.**

R1 - If the value is positive, contains the address of the SDWA that describes the current ABEND that (Note that for ESTAEX or ARR mode, the SDWAPARM field points to an 8-byte field containing the address and ALET of the DBCPARM control block. For other modes, such as ESTAE or ESTAI, the SDWAPARM field points directly to the DBCPARM block.)
If the value is negative, contains the negated address of the DBCPARM control block. (This can occur when there is no SDWA available)

R2 - Contains one of the following exit request codes:

0 - Initialization Exit

1 - Resumption Exit

2 - Termination Exit

3 - Trace User SVC Exit
4 - User Commands Exit
5 - User Register Stack Exit

R13 - Points to a standard 72-byte register save area.
R14 - Points to a return address in Z/XDC.
R15 - Points to the Miscellaneous Exits Interface routine's entry address.
auth - Z/XDC may be either authorized or non-authorized when it calls the interface routine.
super - If authorized, then Z/XDC will be running in supervisor state and key 0.
prob - If non-authorized, then Z/XDC will be running in problem state and jobstep key (usually key 8).
other - Other input states and modes:
PASID=SASID=HASID
Primary ASC-mode
31-bit addressing mode
Enabled, no locks held

The above inputs are set for all calls made by Z/XDC to the Exits Interface, regardless of which specific exit Z/XDC wants the routine to call. Various other registers and DBCPARM block fields may contain values as required by the specific exit that Z/XDC is calling.

OUTPUTS: When the Miscellaneous Exits Interface routine (or any of the exits included with the interface) returns to Z/XDC, it must restore all registers, states, and modes **except** the following:
- **R15:** If the called exit does not exist, then R15 must be set to **4.**
- Otherwise, R15 must be set **according to the needs** of the called exit (which may include R15=4).

Return must be made to **0(,R14).**

For additional information, read the commentary located in the following:
- The **SRCXITSR** member of the **XDCASM** library contains a sample Exits Interface routine.
- The **#DBCPARM** member of the **XDCMACS** library contains a mapping macro for the DBCPARM block.

The factory default names for these libraries are hlq.XDCV31.XDCASM and hlq.XDCV31.XDCMACS, respectively. Ask your Systems Programmer for the libraries' actual names at your Data Center.

Help EXIts RESume

Under construction. Meanwhile, for more information, please:

- Refer to commentary located in the **#DBCPARM** macro located in the XDCMACS library.

(The library's factory default name is **hlq.XDCV31.XDCMACS.** Ask your Systems Programmer for its actual name at your Data Center.)

- Contact ColeSoft. (See **HELP SUPPORT CONTACTUS** for how to do so.)

Help EXIts Termin8

Under construction. Meanwhile, for more information, please:

- Refer to commentary located in the **#DBCPARM** macro located in the XDCMACS library.

  (The library's factory default name is **hlq.XDCV31.XDCMACS.** Ask your Systems Programmer for its actual name at your Data Center.)

- Contact ColeSoft. (See **HELP SUPPORT CONTACTUS** for how to do so.)

Help EXIts UCi

Under construction. Meanwhile, for more information, please:

- Refer to commentary located in the **#DBCPARM** macro located in the XDCMACS library.

  (The library's factory default name is **hlq.XDCV31.XDCMACS.** Ask your Systems Programmer for its actual name at your Data Center.)

- The **XDCONLHG** program makes use of the UCI exit point. The source for this program can be found the **SRCONLHG** member of the XDCASM library. (The factory default library name is **hlq.XDCV31.XDCASM.)** Commentary about and examples of using the UCI can be found therein.

- See **HELP USERINTERFACES UCI** for more information about User Communications Exits.

Help EXIts USErcmds

Z/XDC calls the User Commands exit whenever the user issues a command that neither Z/XDC nor its Fullscreen Support recognizes.

Z/XDC provides to the exit interfaces for several of its internal routines to do such things as display messages, read storage, protect against expected ABENDs, etc. The exit may then either process or reject the command.

Z/XDC calls the User Commands exit via a "Miscellaneous Exits Interface" routine that the user may also have to write. For more information, see HELP EXITS MISCXITS.

INPUTS: The following register settings and DBCPARM block fields are provided by Z/XDC specifically for the User Commands exit. This is **in addition** to those inputs that are documented in HELP EXITS MISCXITS.

DBCPCMDA - Points to the command string to be parsed. The end of the string is delimited by a byte containing X'00'.

DBCPOPER - Points into the command string to the first operand, if any, following the command verb. If no operand is present, then this field points to the zeroed byte delimiting the end of the command string.

DBCPASCB - Points to the ASCB for Z/XDC's currently targeted address space. If Z/XDC is running in Foreign Address Space Mode, then this will be the ASCB for that address space; otherwise, it will be the ASCB for the Home Address Space within which Z/XDC is running. Be aware that this may or may not be the retry level Primary Address Space when debugging a stacking PC routine.

DBCPAWAS - If an ABEND occurs within this User Commands exit while ABEND protection is in effect, then after such an ABEND, this field will contain the ABEND code that occurred.

DBCPWRKA - Points to a 256-byte temporary work area available for use by the exit routine. This area is cleared to zeros prior to each call from Z/XDC to the exit, so this field cannot be used to save information across calls. (The DBCXITSD field can be used for that purpose.)

DBCPCMDT - Contains a code that identifies the category of command being passed from Z/XDC:
1 - A DELETE command.
2 - A LIST command.
3 - A SET command.
4 - A PRINT command.
5 - A DEFINE command.
6 - A TERMINATE command.
0 - Other.

DBCPUSDE - Points to User State Data control blocks describing the user program's error level state data (PSW, registers, cross memory environment, etc.). The control blocks are named the USD and the USDX. They are mapped by the #DBCUSD macro. Warning: The error level state data **must not** be changed by the exit.

DBCPUSDR - Points to User State Data control blocks describing the user program's retry level state data (PSW, registers, cross memory environment, etc.). The control blocks are named the USD and the USDX. They are mapped by the #DBCUSD macro. Note: The retry level state data **may** be changed by the exit.

DBCPOFL1 - This is a flag byte. It contains the following flags:
POF1ENER - This flag indicates whether (=1) or not (=0) the error level and retry level environments are different.
POF1XVAL - This flag indicates whether (=1) or not (=0) the DBCPPROD and DBCPENV fields are valid. (If this flag is not set the fields should be treated as if they contain 0).

DBCPPROD - The 'product' that is active. There are equates that describe the specific products that can be active Note that this field is only valid if the flag in DBCPOFL1 (DBCPXVAL) is set. If the flag is not set, this field should be regarded as containing 0 (DBCPPROD_STD).

DBCPENV - The 'environment' that is active in the case of the standard product (the value in DBCPPROD is DBCPPROD_STD). Note that this field is only valid if the flag in DBCPOFL1 (DBCPXVAL) is set. If the flag is not set, this field should be regarded as containing 0 (DBCPENV_HOST).

DBCPWTOA - Points to a message display service routine.

DBCPRMEM - Points to a storage extraction service routine.

DBCPWMEM - Points to a storage zapping service routine.

DBCPLIFO - Points to a Z/XDC command stacking service routine (LIFO order).

DBCPFIFO - Points to a Z/XDC command stacking service routine (FIFO order).

DBCPPROT - Points to a service routine that establishes temporary protection against specified ABENDs.

DBCPUPRO - Points to a service routine that terminates temporary ABEND protection.

DBCPFSCR - Points to a service routine that notifies Z/XDC to fully repaint its fullscreen display.

DBCPKWDS - Points to a keyword validity checking service routine.

DBCPARSE - Points to a service routine that parses address expressions and returns the resulting address, ASID, and ALET.

RMEMMAX - Is an equate that defines the maximum amount of storage that the DBCPRMEM routine can extract from a Target Address Space.

The interfaces to the service routines are more fully documented in commentary located in the #DBCPARM member of the XDCMACS library. (The library's factory default name is hlq.XDCV31.XDCMACS. Ask your Systems Programmer for its actual name at your Data Center.)

Subtopics Index

A sample User Commands exit has been written and is available in both source and executable form. This exit implements a **LIST TIOT** command. To view information about this sample exit, Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

SRCXUCMD - A sample user commands exit routine.

Help EXIts USErcmds Srcxucmd

A sample User Commands Exit has been written and is available in both source and executable form. The executable form is in a load module named **XDCXITS** and can be found in the XDCLINK library. (The factory default name is hlq.XDCV31.XDCLINK. Ask your Systems Programmer for the library's actual name at your Data Center.)

XDCXITS is constructed from several csects. The source code for three of those csects is published. They are:

- A sample Miscellaneous Exits Router named **SRCXITSR**
- A sample User Commands Exit named **SRCXUCMD**
- And a sample User Register Stack Exit named **SRCXUREG**

The source code for SRCXITSR, SRCXUCMD and SRCXUREG can be found in the XDCASM library. (The factory default name is hlq.XDCV31.XDCASM.)

SRCXUCMD implements a **LIST TIOT** command that can be used to display the dataset allocations for any task in any accessible address space.

If you wish to write support for your own user commands, then SRCXUCMD is a good, comprehensive model to follow.

If you wish to write your own user commands and also **retain** our implementation of the LIST TIOT command, you can do that too. SRCXUCMD has logic that looks for a separate, user written program named **USRXUCMD.** If present, then SRCXUCMD will call that program at a point where it has determined that it does not recognize the given command, (i.e. it is not LIST TIOT or UCTEST). Then USRXUCMD can perform its own parsing and return to SRCXUCMD with a success/fail return code that SRCXUCMD will forward back to Z/XDC proper.

To install a USRXUCMD routine, you need to write one, give it an entry point named USRXUCMD, and link edit it into the XDCXITS load module. SRCXUCMD has a **WXTRN** for USRXUCMD, so if you provide the routine, SRCXUCMD will call it, and if you don't provide one, (guess what...) SRCXUCMD won't call it.

USRXUCMD must be written to run in the following environment with the following inputs:

- The routine will have to be link-edited into the XDCXITS load module.
- Its entry point name must be USRXUCMD.
- It will always run in task mode (never SRB mode).
- The hardware environment will be as follows:
    - PASID = SASID = HASID.
    - State = Supervisor or problem.
    - Locks = none.
    - Key = 0 if supervisor state.
    - Key = TCBPKF (usually =8) if problem state.
    - ASC-MODE = Primary.
    - AMODE = 31.
- The registers will be as follows:
    - **R0** will contain a copy of **DBCXITSD.**
    - **R1** will point to the **SDWA.**
    - **R13** will point to a standard, 72-byte **RSA.**
    - **R14** will point to a **return address.**
    - **R15** will point to **USRXUCMD.**

    - Both the SDWA, the DBCPARM block, and the register save area will be located

in the **home** address space.

Upon return:
- **R15** must contain one of the following return codes:
  0 - The command was recognized and processed successfully.
  4 - The command was not recognized.
  8 - The command was recognized but its processing failed.
- All other registers must be restored.
- All hardware modes and states must be restored to those described above.

The process of making the XDCXITS load module available for use by Z/XDC is discussed in HELP EXITS MISCXITS.

Help EXIts USVctrce

Under construction. Meanwhile, for more information, please:

- Refer to commentary located in the **#DBCPARM** macro located in the XDCMACS library.

  (The library's factory default name is **hlq.XDCV31.XDCMACS.** Ask your Systems Programmer for its actual name at your Data Center.)

- Contact ColeSoft. (See **HELP SUPPORT CONTACTUS** for how to do so.)

Help EXIts REGstack

The c/XDC component of Z/XDC calls the **User Register Stack Exit** during variable discovery for Metal C language environments. The Exit's purpose is to examine the general registers from a given level of subroutine execution and return with the general registers that belong to that subroutine's **caller.**

A user **must** provide this Exit when the Metal C programs he is working with **do not use** standard register save area linkage.

Upon return, the Exit must either provide a copy of the data from the requested set of general registers, or it must indicate that the given subroutine linkage structures were not understood.

Available Sample Code

Z/XDC calls the Register Stack Exit via its **Miscellaneous Exits Interface.** Sample versions of **both** the Register Stack Exit and the Miscellaneous Exits Interface are provided in both source and load module form in the **XDCASM** and **XDCLINK** libraries, respectively. The factory default name for the XDCASM library is hlq.XDCV31.XDCASM. For the XDCLINK library, it's hlq.XDCV31.XDCLINK. Ask your Systems Programmer for the library's actual name at your Data Center.

The sample Miscellaneous Exits Interface can probably be used as is. The sample Register Stack Exit, on the other hand, will most certainly have to be customized. For more information, see HELP EXITS MISCXITS.

The DBCPARM Control Block: General Features

DBCPARM contains pointers to and interface data for several of Z/XDC's internal services so that they can be used directly by the various exits. These services include subroutines for:

- Message display
- User storage access
- ABEND protection management
- Etc.

These pointers and interfaces are described in detail by commentary located in the **#DBCPARM** macro, located in the **XDCMACS** library. For more information, see HELP MACROS #DBCPARM.

DBCPARM Fields Specifically for the Register Stack Exit

In addition to the common interface structures just mentioned, **c/XDC** provides data specifically for use by the Register Stack Exit.

DBCPREGI - Points to a control block called the **Extended User State Data** block. This is where c/XDC places the register contents for the subroutine level that he is currently examining.

The mapping dsect's name is **USDX,** and it is generated by the **#DBCUSD** macro. That macro can be found in the **XDCMACS** library.

The following USDX fields are filled in:

UGR_LO - contains the low halves of all 16 general registers ordered from R0 thru R15.

UGR_HI - contains the high halves of all 16 general registers ordered from RH0 thru RH15.

UAR - contains all 16 access registers ordered from AR0 thru AR15.

- All **other** USDX fields can be ignored.
- When a register type is **unused** in your Metal C program, its values are all zero'd.

DBCPREGO - Points to another USDX-mapped area where your Exit must place the register values for the given subroutine's **caller,** i.e. those register values that would be restored when said subroutine returns to its caller.

It is expected that your Exit will be able to find those register values by inspecting and using the registers provided by **DBCPREGI.**

IMPORTANT! When accessing your program's data (such as register stacks), you **must not** do so directly. Instead, you must use the storage access service routine pointed by **DBCPRMEM.**

DBCPADDM - When available, points to the start of the module or program object that contains the subroutine that is currently being examined by c/XDC.

DBCPADDC - When available, points to the start of the csect that contains the subroutine that is currently being examined by c/XDC.

DBCPADDF - When available, points to the start of the subroutine that is currently being examined by c/XDC.

DBCPNAMM - When available, points to the null-delimited name of the module or program object that contains the subroutine that is currently being examined by c/XDC.

DBCPNAMC - When available, points to the null-delimited name of the csect that contains the subroutine that is currently being examined by c/XDC.

DBCPNAMF - When available, points to the name of the subroutine that is currently being examined by c/XDC. In this case, the name is not null-delimited. Instead, it is in a string-length format **[AL1(L'name),C'name'].**

DBCPASCB - Points to the ASCB for Z/XDC's currently targeted address space. If Z/XDC is running in **Foreign Address Space Mode,** then this will be the ASCB for that address space; otherwise, it will be the ASCB for the Home Address Space within which Z/XDC is running. Be aware that this may or may not be the retry level Primary Address Space when debugging a stacking PC routine.

DBCPAWAS - If an ABEND occurs within this Register Stack Exit while ABEND protection is in effect, then after such an ABEND, this field will contain the ABEND code that occurred.

DBCPWRKA - Points to a 256-byte **temporary** work area available for scratch use by the Exit routine. This area is cleared to zeros prior to each call from Z/XDC to the Exit, so this field cannot be used to save information across calls. (Use the **DBCXITSD** field for that purpose.)

DBCPWTOA - Points to a message display service routine.

DBCPRMEM - Points to a storage extraction service routine.

DBCPWMEM - Points to a storage zapping service routine.

DBCPPROT - Points to a service routine that establishes temporary protection against specified ABENDs.

DBCPUPRO - Points to a service routine that terminates temporary ABEND protection.

DBCPFSCR - Points to a service routine that notifies Z/XDC to fully repaint its fullscreen display.

DBCPKWDS - Points to a keyword validity checking service routine.

DBCPARSE - Points to a service routine that parses a text string containing an address expressions and returns the resulting address, ASID, and ALET.

RMEMMAX - Is an equate that defines the maximum amount of storage that the DBCPRMEM routine can extract from a Target Address Space.

The interfaces to the service routines are more fully documented in commentary located in the **#DBCPARM** member of the the **XDCMACS** library. (The library's factory default name is hlq.XDCV31.XDCMACS. Ask your Systems Programmer for its actual name at your Data Center.)

Register and State Inputs

When Z/XDC calls the Register Stack Exit, the registers and environment are set as follows:

asid - The Primary and Secondary Address Spaces are both set to being the Home Address Space. This is true even when the environment being debugged is a cross memory environment.

R0 - Contains a copy of the **DBCXITSD** field of a **DBCPARM** block. Initially, this field is zero'd, but The Register Stack Exit routine may change the contents of this field at any time for any purpose. Any such change will be preserved by Z/XDC from one call to the next. For more information, see HELP MACROS #DBCPARM.

R1 - Points to the SDWA control block that describes the current ABEND that caused Z/XDC to receive control. (Note that for ESTAEX or ARR mode, the SDWAPARM field points to an 8-byte field containing the address and ALET of the DBCPARM control block. For other modes, such as ESTAE or ESTAI, the SDWAPARM field points directly to the DBCPARM block.)

R13 - Points to a standard 72-byte register save area.

R14 - Points to a return address in Z/XDC.

R15 - Points to the Register Stack Exit's entry address.

auth - Z/XDC may be either authorized or non-authorized when it calls the interface routine.

super - If authorized, then Z/XDC will be running in supervisor state and key 0.

prob - If non-authorized, then Z/XDC will be running in problem state and jobstep key (usually key 8).

other - Other input states and modes:
PASID=SASID=HASID
Primary ASC-mode
31-bit addressing mode
Enabled, no locks held

Subtopics Index

For additional information, type an **H** at the left to select directly, or use **HELP**

***NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

SRCXUREG - A sample user Register Stack Exit routine.

Help EXIts REGstack Srcxureg

A sample User Register Stack exit has been written and is available in both source and executable form. The executable form is in a load module named XDCXITS and can be found in the XDCLINK library. (The factory default name is hlq.XDCV31.XDCLINK. Ask your Systems Programmer for the library's actual name at your Data Center.)

XDCXITS is constructed from several csects. The source code for three of those csects is published. They are a sample Miscellaneous Exits router named **SRCXITSR** , a sample User Commands exit named **SRCXUCMD.** , and a sample User Register Stack exit named **SRCXUREG.** The source code for SRCXITSR, SRCXUCMD, and SRCXUREG can be found in the XDCASM library. (The factory default name is hlq.XDCV31.XDCASM.)

SRCXUREG provides support for traversing standard z/OS type-1 save area chains during c/XDC variable discovery for metal C language environments.

If you wish to write support for other metal C linkages, then SRCXUREG is a good, comprehensive model to follow.

DBCXUREG must be written to run in the following environment with the following inputs:

- The routine will have to be linkedited into the XDCXITS load module.
- Its entry point name must be XITUREG.
- It will always run in task mode (never SRB mode).
- The hardware environment will be as follows:
  - PASID = SASID = HASID.
  - State = Supervisor or problem.
  - Locks = none.
  - Key = 0 if supervisor state.
  - Key = TCBPKF (usually =8) if problem state.
  - ASC-MODE = Primary.
  - AMODE = 31.
- The registers will be as follows:
  - **R0** will contain a copy of **DBCXITSD.**
  - **R1** will point to the **SDWA.**
  - **R13** will point to a standard, 72-byte **RSA.**
  - **R14** will point to a **return address.**
  - **R15** will point to the entry address **(XITUREG).**

  - Both the SDWA, the DBCPARM block, and the register save area will be located in the home address space.

Upon return:
- **R15** must contain one of the following return codes:
  0 - The "current" subroutine environment was recognized. The output area was updated with the registers of the "current" subroutine's caller.
  4 - The "current" subroutine environment was not recognized. c/XDC should attempt to locate the "current" subroutine's caller itself.

8 - The "current" subroutine environment was recognized. The "current" subroutine is the "main" routine, and does not have a "caller".

- All other registers must be restored.
- All hardware modes and states must be restored to those described above.

The process of making the XDCXITS load module available for use by c/XDC is discussed in HELP EXITS MISCXITS.

Help FULlscreen

Z/XDC writes its displays to 3270-type fullscreen workstations. Once upon a time, these workstations were hardware terminals, but these days they are created by emulating software that you run on your PC. Perhaps the most common are IBM's **PCOMM** and Tom Brennan's **VISTA.** For more information, see **HELP FULLSCREEN TERMINALS GEOMETRIES.**

Z/XDC's 3270 support is implemented by a component called Terminal Fullscreen Support, also known as **TFS.** When TFS is present and turned on, Z/XDC presents the user with an interactive fullscreen display. When TFS is absent or turned off, Z/XDC falls back to using a linemode communication style. TFS can be turned off and on with the **SET TFS** command.

Z/XDC's TFS is, perhaps, the most feature rich exploiter of 3270 architecture you will ever see. It fully supports large display geometries, and nearly every field displayed is reactive in multiple useful ways!

- **Color** is used to good effect for characterizing displayed fields and other information.
- Most displayed lines accept **shortcuts** that vary according to the nature of the information displayed.
- **Point-and-Shoot** support can be used **on data** to follow links through storage.
- **Point-and-Shoot** can also be used **on machine instructions** to follow their storage pointers no matter what kind of pointers an instruction uses!
- Both data and machine instructions can be **zapped** by overtyping **hex** values, **text** values and even machine code **mnemonics!**
- The display screen can be subdivided into **multiple subwindows** that can watch whatever storage, registers, data fields, control blocks or reports you like.

Z/XDC's fullscreen display is one of the features that gives Z/XDC its extraordinary power. It was originally designed and coded by a then young Michael Lewis. While many of the details of the design have been tweaked and improved over the decades; overall, its original structure has remained quite durable.

Michael is quite proud of his work, and rightly so!

More Information

Ok, let's get into the weeds, shall we? For detailed information, type an **H** at the left below to select directly, or use **HELP *NEXT (F11)** to proceed sequentially.

Use **HELP *FORWARD (F5)** to skip.

KEYWORD	DESCRIPTION
TERMINALS	- Setting up workstation programs for use with Z/XDC.
TITLE	- Information available from Z/XDC's screen title line.
FSINPUTS	- Input fields and methods available with fullscreen displays.
DISPLAYERRS	- Recovering from repeated "screen erasure" errors.
ISPF	- ISPF compatibility and independence.
LOGGING	- Session logging.
NOTES	- A note pad for recording debugging related remarks.
PFKEYS	- Program function key defaults and management.
RETRIEVE	- Command retrieval service.
SCROLLING	- Scrolling through your session log and scrolling through storage.
WINDOWS	- Multiple user definable display areas.

Help FULlscreen TErminals

Z/XDC has the ability to take full advantage of all that the 327x terminal architecture has to offer. It has full support for showing color displays, using hilighting, and making use of of large display geometries. Unfortunately, oftentimes users are trying to use Z/XDC in environments where terminal emulators have not been well installed, and so they have to cope with unnecessary limitations. In other words, if you're using a terminal emulation that displays only 25 lines, or if you're using a terminal that shows only two colors (bright and dim), then you're working with an unnecessary handicap, and you're not going to be able to use Z/XDC to anything that's even close to its fullest capabilities.

Subtopics Index

For information about activating various display capabilities, select the following topics. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

GEOMETRIES	- Setting up large display geometries.
ISPFSETTINGS	- ISPF settings needed for using large screen geometries.
COLORS	- Enabling color terminals.
BINDIMAGES	- If you're having problems getting large displays (or color displays) to work, you probably have an incorrect BIND image.
PROBLEMS	- For information about terminal communications problems in general.

Help FULlscreen TErminals Geometries

Z/XDC works best on terminals with large display geometries. To avoid line truncation on the right, the display should be 136 characters wide. And of course the more rows you can set up, the better.

By now (2010), Z/XDC has had large Display Geometry support for over a decade! So in recent years I have been writing more and more **LIST** commands that produce wide reports.

Command Dialogs

In addition, in Z/XDC release z1.12, we introduced **Command Dialogs** to make it easier to use some of Z/XDC's more complex and arcane commands. Command Dialogs are panels that contain both captioned fill-in fields and explanatory information. All in all, these panels can be **quite large,** so they do not display well in small geometries. So if you want make the best advantage of Command Dialogs, then you will have to use a display that is maybe 50 or 60 (or more) rows high. In addition, we recommend that your display also be **136 columns** wide (or more).

Large Display Geometry Considerations

The 327x protocol supports displays up to 255 rows by 255 columns, but there are several factors that impose smaller limits:

- **TSO:** As far as I know, TSO does not have a documented display size limitation; however, I have found that when 128 or more display rows are configured, errors begin to occur in TSO's management of its displayed messages.

- **ISPF:** I have tested up to **72-row** displays without problems. As of z/OS R2.5, the ISPF's max supported width is **160 columns.**

  When its limits are exceeded, ISPF reverts to a 24x80 display. **[Ughhh!]**

- **IPCS:** This program, when it displays storage, will display 32 bytes worth per line, but only when the line length is 136 characters or wider. (For smaller line lengths, IPCS will display only 16 bytes worth of storage per line.)

- **Z/XDC:** For most displayed output, the longest message that Z/XDC will construct is 136 characters. (But many messages will flow to fill whatever display width is provided.)

  Regarding overall display size, Z/XDC does not have any explicit limitations. In particular, Z/XDC's displays are not affected by ISPF's limitations. For example, if you set your terminal's geometry to 62x161):
  - ISPF will revert to a 24x80 display.
  - But Z/XDC will go ahead with 62x161 displays.

  However, Z/XDC's display buffer size is 32K bytes. Buffer overflows can occur unpredictably but with increasing likelihood as the total screen area size approaches and then exceeds 32 thousand characters (approximately 181x182). So I'll not guarantee support for any screen geometry whose row-by-column product gets too close to this 32K limit.

- **Your Eyes:** As emulated display geometries increase, the font needed to display

the data becomes smaller and smaller. Once a font falls below about 6 points or so, it becomes unreadable.

On my desktop, I have a 21" monitor. I find that 62 rows is about the largest geometry that my eyes can see clearly. Your eyes, of course, may be better than mine. They may be worse.

All in all, it seems that for the best results, you should try to set up your terminal emulation to be **62x160** (high-by-wide).

Session Profile Support for Large Displays

If you do not routinely use large display geometries, I would strongly recommend that you start doing so, **even if** that means changing out your PC's monitor for a larger one. **Even if** you have to spend your own money to do so! The improvement in your productivity, both inside and outside of Z/XDC, will be that significant!

If you do use large display geometries, then for Z/XDC you will want to establish a session profile that will let you take advantage of the extra real estate. As it happens, the Z/XDC Product Package includes an optional session profile that is specifically suitable for large display geometries.

- The profile is named **WIDE.**
- Its SMP/E TLIB library is **XDCTLIB.**
- The factory default name for XDCTLIB is **hlq.XDCV31.XDCTLIB.**
- Ask your Systems Programmer what this library's actual name is at your data center.
- Its PDS member name (within XDCTLIB) is **TFSWIDE.**

When Z/XDC was installed, **TFSWIDE** should have been copied into one of your System's **ISPF ISPTLIB** libraries.

To load the **WIDE** profile and make it your permanent personal profile, issue the following commands:

PROF READ WIDE TFS
This finds and loads a profile named **WIDE** from a PDS member named **TFSWIDE.**

PROF SAVE XDC
This renames your profile to **XDC** and saves it as your permanent default personal profile. It will be saved in your ISPPROF library in the member named **xxxXDC31.**

LIST PROF
This shows detailed information about your profile's name, library, version etc.

PROF
This starts up the Profile Menuing System. This a series of panels that display and allow you to change almost all profiled settings.

The following are the differences between the **WIDE** profile and the Factory Default profile.

SET FORMAT WIDE
This causes several of Z/XDC's various display producing commands to create 32-byte wide (instead of 16-byte wide) displays.

L PSW;L EPSW;L RWREGS;;L AREGS

The secondary display window is a few lines larger, and it contains a more comprehensive set of displays.

Workstation Support for Large Displays

There are, of course, a very large number of workstation emulation programs that you can run on your PC. Some of them support flexible screen geometries, others do not. Here are the ones with which I've had personal experience:

- **OpenText's Host Explorer**
  http://connectivity.opentext.com/products/terminal-emulation.aspx

  Version 7.1 of this product supports arbitrary geometries up to 72 rows by 200 columns. To change your geometry, proceed as follows:
  - First, disconnect your emulated terminal.
  - Then go to the "Options" menu, and select "Edit Session Profile...".
  - Then open the "Terminal" folder and select the "3270" tab. (If everything is grayed out, then you forgot to disconnect the terminal.)
  - From the "3270 Type" dropdown box, select "3279".
  - From the "3270 Model" dropdown box, select "Custom". This will cause the "Rows" and "Columns" fields to be ungrayed.
  - Fill in the geometry that you desire, and then click on the "OK" button.
  - To make your change permanent, go to the "File" menu, and select "Save Session Profile...".

- **IBM's Personal Communications (PCOMM)**
  www-01.ibm.com/software/network/pcomm

  Version 5.5 of this product supports only standard geometries via its setup dialog boxes, but if you're willing to manually change a terminal's underlying definition file, you can cause PCOMM to emulate any reasonable display size. To do so, proceed as follows:
  - First, you will need to set the terminal to a nondefault size. (Doing so will cause PCOMM to add a "ScreenSize=..." statement to the terminal's definition file.) To set a nondefault size:
    - Start a terminal session.
    - Click on the "Communication" menu, and select "Configure...".
    - Click on "Session Parameters".
    - In the "Screen Size:" dropdown box, select any geometry other than 24x80. ("43x80" for example.)
    - Click OK to close the "Session Parameters" dialog box.
    - Click OK to close the "Customize Communication" dialog box.
    - You may get a message box saying, "PCSCC041 - Because you have changed the configuration, communication will be terminated if you proceed. Are you sure?". If you do see this message, click OK.
    - At the session window, open the "File" menu, and click on "Save" to save the configuration change.
    - Close the session window.
  - Now you will have to locate the terminal's definition file and manually change it using a text editor (NOTEPAD.EXE for example). To find the file:
    - Open the session again.
    - At the session window, open the "File" menu, and click on "Save As...". This will open the "Save Workstation Profile as" dialog box.
    - Near the top of the box, there is a dropdown list captioned at its left as "Save in:" At the right of the list's window, there is a down

arrowhead. Single click on the arrowhead to expand the Folder Location Display. Probably, it will show something like this:

```
My Computer
  harddrivename (C:)
    Document and Settings
      windowsuserid
        Application Data
          IBM
           Personal Communications
```

  - Also, the "File Name:" box will show the definition file's actual name. It will have a ".ws" suffix.
  - Anyway, write down the path and file name.
  - Then click on "Cancel" to close the dialog box without saving the definition again.
  - Then close the session window again.
- Now start up NOTEPAD.EXE, or your own favorite text editor. Do NOT use a word processing program unless you really really know what you're doing.
- Navigate to the session's definition file, and open it.
- Look for the statement that reads something like, "ScreenSize=43x80".
- Replace the statement's operand ("43x80" in this case) with the dimensions that you desire (62x136) for example). Specify the rows first and the columns second.
- Save the file back, and close your editor.
- Finally, go back to PCOMM and start up the session.

Whenever you first start up a TSO session, the logon process will always use 24x80 displays. However, once you're past the logon screen, TSO will switch to using whatever dimensions you've specified.

Just today (September 30, 2010) I discovered an IBM written article describing essentially the same method that I've outlined above for coercing PCOMM into producing a large geometry display. The article can be found at www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/TD102151. That article was written in 2008. I am **astonished** that PCOMM setup dialogs **still** do not provide a way to setup large geometries! Come on guys! Your competitors have been doing it for over 10 years! (Well... Maybe not Attachmate...)

- **Attachmate's EXTRA!**
  http://www.attachmate.com/en-US/Products/Products.htm

  The "Extra! Enterprise 2000" version of this product supports **only** standard 327x display geometries (24x80, 32x80, 43x80, and 27x132). It does **not** support. custom geometries!

  There is a newer version of EXTRA! called "myEXTRA! Smart Connector for 3270 & 5250", but I do not have personal experience it.

There are, of course, many other workstation emulation programs with which I do NOT have personal experience. Some of them are:

- **HOB Electronic's HOBLink Terminal Edition**
  http://www.hobsoft.com/produkte/classemu.jsp

- **Micro Focus's Rumba**
  http://www.microfocus.com/products/RUMBA

- **Tom Brennan Software's Vista tn3270** (shareware). I've been told that this

product supports arbitrary display geometries.
http://www.tombrennansoftware.com

Finally if, after you've configured your workstation program, you still can't get large displays to work, you've probably got a **BIND image problem.** See **HELP FULLSCREEN TERMINALS BINDIMAGES** for more information.

Help FULlscreen TErminals Ispfsettings

Setting up ISPF to make use of large geometry display screens is pretty easy, just so long as you know the secret settings... (I.E. IBM's names for the necessary settings are not real clear.) Here's what you need to do:

- Navigate to the **ISPF Settings** panel (=0).
- Scroll down (PF key F8), if necessary, to bring the **Terminal Characteristics** portion of the panel into full view.
- Set **Screen format** to **3** ("Max").
- Set **Terminal Type** also to **3** ("3278").
- Press **ENTER** and ISPF should immediately switch to the larger display.

If the screen does not change, then you may have one or two other problems to overcome:

- You may have to tell your Workstation Program (PCOMM or Attachmate or whatever) to send large displays. See **HELP FULLSCREEN TERMINALS GEOMETRIES** for more information.
- Or you may have to tell VTAM to accept large screen displays from your Workstation Program. See **HELP FULLSCREEN TERMINALS BINDIMAGES** for more information.

Once you get everything else right, when you start Z/XDC, it may still revert to a 25x80 display. To correct this:

- Issue **SET SECONDARYSIZE** to cause Z/XDC to generate large displays.
- Issue **PROFILE SAVE** to to permanently save this change into your personal default session profile.

You may also want to read **HELP FULLSCREEN TERMINALS PROBLEMS** for more information.

Help FULlscreen TErminals Colors

These days, all workstation emulation programs offer full support of color displays. in fact, most (if not all) also permit you to remap the colors that the mainframe sends into any other colors of your choice. This remapping is one capability that I find very valuable. Maybe it's just me, but I think that the 3270 default blue color is just about impossible to see against black. It's just too dark. So I always remap that blue into another blue that is several shades lighter.

Anyway, if you find that your terminal emulator is not giving you multi-color support, then it's probably not the fault of the emulator. Somewhere, an incorrect setup choice has been made. Usually, this will be due to one or the other of two causes. The simple one first:

- **EABs (Extended Attribute Bytes)**
  Extended Attribute Bytes are the part of the 327x display protocols that are needed for complete color support (among other things). Generally, somewhere within a workstation program's setup dialog or options dialog or preferences dialog, there will be a check box that indicates whether or not you want your emulated terminal to support EABs. Be sure that this box is checked. It usually will be clustered with other settings relating either to the display's definition or to the terminal's definition. (Note, PCOMM doesn't seem to have this setting. Its EAB support is apparently always enabled.)

- **BIND image**
  It's rather complicated, but a "BIND image" is a VTAM control block that is associated with a user terminal's connection to TSO. It resides on the mainframe side of the connection (not in the workstation program), and it identifies to programming what the capabilities of the connected terminal are.

  There generally is nothing that prevents a BIND image from lying! For details, see **HELP FULLSCREEN TERMINALS BINDIMAGES.**

Help FULlscreen TErminals Bindimages

A **BIND image** is a VTAM control block that provides information about the capabilities of a terminal when it is connected to the mainframe. BIND images reside within VTAM (in MVS), not at the terminal, and they are created and modified by VTAM Systems Programmers. Unfortunately, VTAM is pretty complicated, and many data centers do not have Systems Programmers that know very much about VTAM.

There are two important things that a BIND image tells fullscreen displaying programs (like TSO and like Z/XDC) about a terminal. They are:

- Whether or not the terminal supports Extended Attribute Bytes (for color support, extending hilighting, and such).

- Whether or not the terminal can be QUERY'd, i.e. asked what its capabilities and display dimensions are.

If, according to the BIND image, the terminal cannot be queried, then the BIND image itself will contain the terminal's supposed display dimensions.

There's nothing that prevents a BIND image from lying. In particular, it is

frequently the case that the default BIND image for a TSO terminal connection will:
(a) - Claim that the terminal does not support QUERY when it does.
(b) - Claim that the terminal supports only a 24x80 display when in fact it can support larger geometries.

If you are the victim of a limited BIND image, you will have to ask your Systems Programmer to fix the problem. Meanwhile, you can override the default BIND image on the initial TSO LOGON command by using the LOGMODE=name operand. Example:
LOGON userid LOGMODE=D4C32XX3

There is no "standard" name for the desirable BIND images; however, there is a "sample" log mode table that's distributed with MVS, and most systems automatically use it. The sample log mode table's name is ISTINCLM, and its source code is distributed in SYS1.SAMPLIB.

For BIND images, a log mode table entry's "PSERVIC string" is key. At ColeSoft, our log mode table entry that supports query has **PSERVIC=X'028000000000000000000300'.**

There are several log mode table entries defined in the sample table (ISTINCLM) that would work well with Z/XDC. Perhaps the "preferred" one is D4C32XX3. (A couple of others that also seem to work are D4A32XX3 and D4B32XX3.)

So to force your TSO connection to use the log mode table entry named D4C32XX3, you have to specify it on your TSO initial LOGON command as follows: **LOGON userid LOGMODE=D4C32XX3**

If your terminal session has a BIND image that you think ought to work but doesn't, then you can ask us (ColeSoft) to troubleshoot the problem, and we will probably ask you to provide to us a terminal communications trace. To obtain such a trace, just allocate a file to JESn spool using the ddname XDCTRACE (or xxxTRACE if Z/XDC is running under the clone name "xxx"). Example: "//XDCTRACE DD SYSOUT=*". For our contact information, see **HELP SUPPORT CONTACTUS.**

Help FULlscreen TErminals Problems

If you are having problems getting large display geometries to work, first review the list of things to check found in **HELP FULLSCREEN TERMINALS ISPFSETTINGS.**

If you still are having problems, then give us a call. We're pretty good at troubleshooting this stuff. You can find our contact information at **HELP SUPPORT CONTACTUS.** You may call **Level-2 Support** directly.

Also, there's this... Occasionally, a user may experience a strange issue with one 3270 emulator or another. We'll shoot the problem, and if the solution is particularly interesting or unexpected, we will post it here for the benefit of the rest of our customers.

ATTN Key Problems in Tom Brennan's VISTA TN3270 Emulator

One customer, using Vista 1.24, reported that the **ATTN** function (mapped to

his **ESC** key) worked generally, but not while he was using Z/XDC. He emailed both me and Tom Brennan about this, and Brennan came up with the following solution:

> First, this problem can arise only when TN3270E is **UNchecked** in the Options dialogs. The best choice would be to enable TN3270E support.
>
> But if TN3270E=OFF is really what you want, then you will have to find the VISTA.INI file for your session and add one or the other of the following to the file:
>
> ```
> AttentionMethod=1
> AttentionMethod=2
> ```

The customer found that **AttentionMethod=1** resolved his problem.

General Problem Resolution

Z/XDC has at least six substantially different mechanisms for communicating with a user's terminal. They include:

- linemode TPUT/TGETs
- fullscreen TPUT/TGETs
- ISPF Display Services
- VTAM SEND/RECEIVE
- cdf/XDC
- WTO/WTOR to Extended MCS (EMCS) System Operator Consoles

The protocols are varied and complex, so it is possible that problems will be encountered, and when they do, you can, of course, ask us (ColeSoft) to troubleshoot them. For our contact information, see **HELP SUPPORT CONTACTUS.**

Help FULlscreen TItle

Every fullscreen display that Z/XDC emits has a top line screen title that conveys generally useful information about the debugging session. There are three sections in the title line: a left side, a right side and a center section.

For details select the following: Type an **H** at the left below to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

CENTER - The center text provides information about the connection between your debugging session and your terminal, as well as the communications method currently in use.

LEFT - The left side text provides information about nature of the program being debugged: TCB v SRB, authorized or not, target aspace when in FASM mode, etc.

RIGHT - The right side text provides additional information: Your location in the top window's scroll area, when tasks are waiting during multitasking debugging, the aspace in which your debugging session is running... It also is the place where transient messages are displayed reporting warnings of

various sorts elicited by the command you just issued.

Help FULlscreen TItle Center

Every fullscreen display that Z/XDC emits has a top line screen title that conveys generally useful information about the debugging session. There are three sections in the title line: a left side, a right side and a center section. Here, I will describe the **center** section.

The title's center section shows information about the method by which Z/XDC is communicating with your terminal. The possibilities are as follows:

Z/XDC TPUT INTERFACE
This information conveys the following:

- You are using a TSO session to connect to your debugging session which also is running within TSO.
- Your session may or may not be running under ISPF.
- Z/XDC is using fullscreen TPUT/TGETs to paint and read the screen.
- If you are running under ISPF, then you have issued a **SET TPUT** command to avoid using ISPF services.
- Or you are running authorized.
- You may **not** use ISPF's **SPLIT** and **SWAP** commands.
- You may **not** use ISPF's =jump commands **(=6, =X, =3.3, =whatever).**
- All other aspects of your debugging session are identical regardless of whether or not ISPF is in use.

Z/XDC ISPF INTERFACE
This information conveys the following:

- You are using a TSO session to connect to your debugging session which also is running within TSO.
- In fact, your debugging session is running within ISPF.
- Your debugging session is running nonauthorized.
- Z/XDC is using ISPF's Display Services to paint and read the screen.
- You have issued (or defaulted to) a **SET ISPF** command to allow Z/XDC to use ISPF services.
- You **may** use ISPF's **SPLIT** and **SWAP** commands to switch to another program within ISPF without ending your debugging session.
- You **may** use ISPF's =jump commands **(=6, =X, =3.3, =whatever)** to end your session and jump directly to another program within ISPF.
- All other aspects of your debugging session are identical regardless of whether or not ISPF is in use.

XDC-CDF TPUT INTERFACE

This information conveys the following:

- You are running a debugging session outside of TSO. It could be in the batch, in a system task, in a CICS transaction, in pretty much anywhere that is not TSO.

- The debugging session could even be running in a different TSO session if the **//XDCCDF DD DUMMY** allocation is present there.

- You are connected to the session from **option 3** of Z/XDC's Startup Panel in ISPF.
- Your connection interface always runs non-authorized in ISPF.
- Your debugging session may or may not be running authorized.

- You have issued a **SET TPUT** command to avoid using ISPF services.

- You may **not** use ISPF's **SPLIT** and **SWAP** commands.

- You may **not** use ISPF's =jump commands **(=6, =X, =3.3, =whatever).**
- All other aspects of your debugging session are identical regardless of whether or not ISPF is in use.

XDC-CDF ISPF INTERFACE

This information conveys the following:

- You are running a debugging session outside of TSO. It could be in the batch, in a system task, in a CICS transaction, in pretty much anywhere that is not TSO.

- The debugging session could even be running in a different TSO session if the **//XDCCDF DD DUMMY** allocation is present there.

- You are connected to the session from **option 3** of Z/XDC's Startup Panel in ISPF.
- Your connection interface always runs non-authorized in ISPF.
- Your debugging session may or may not be running authorized.

- You have issued (or defaulted to) a **SET ISPF** command to cause your local connection to use ISPF services.

- You **may** use ISPF's **SPLIT** and **SWAP** commands to switch to another program within ISPF without ending your debugging session.

- You **may** use ISPF's =jump commands **(=6, =X, =3.3, =whatever)** to end your session and jump directly to another program within ISPF.
- All other aspects of your debugging session are identical regardless of whether or not ISPF is in use.

XDC-CDF VTAM INTERFACE

This information conveys the following:

- You are running a debugging session outside of TSO. It could be in the batch, in a system task, in a CICS transaction, in pretty much anywhere that is not TSO.

- The debugging session could even be running in a TSO session if the **//XDCCDF DD DUMMY** allocation is present there.

- You are connected to the session via a direct VTAM logon **(LOGON APPLID=XDC** for example).
- Your debugging session may or may not be running authorized.

- You may **not** use ISPF's **SPLIT** and **SWAP** commands.

- You may **not** use ISPF's =jump commands **(=6, =X, =3.3, =whatever)** to end your session and jump directly to another program within ISPF.
- No other aspects of the debugging session are affected.

Built-in Help Topics

The center title reports that you are in a topic display of the Built-in Help.

It also reminds you how to use the **SCANLOG** command to search for text strings within the topic.

Profile Menuing System
When you are displaying panels from the Profile Menuing System the center title identifies which panel you are currently viewing.

Help FULlscreen TItle Left

Every fullscreen display that Z/XDC emits has a top line screen title that conveys generally useful information about the debugging session. There are three sections in the title line: a left side, a right side and a center section. Here, I will describe the **left side** information.

The title's left side shows information about the nature of the program being debugged: TCB v SRB, authorized or not, target aspace when in FASM mode, etc. The possibilities are as follows:

TCB#n RB#n ---
This information indicates that Z/XDC has received control as an ESTAE-type recovery routine and that you are debugging a program that is running in task mode.

The **#n's** indicate:
- Under which task your program is running,
- And under which Request Block your program is running.

The **#n's** correspond to the reports given by the **LIST TASKS** and **LIST RBS** commands.

They also correspond to the automatic equates that those two commands create.

SRB PROXY (AUTH) ---
This information indicates that Z/XDC has received control as an **FRR** (not an ESTAE-type), so you must be trying to debug either an SRB routine or a task mode routine doing something that requires FRR protection.

However, most of what Z/XDC does is not compatible with an FRR execution environment, so the FRR protected routine (task or SRB) has been suspended, and Z/XDC itself is now running in a proxy task.

... (AUTH) ---
When present, this indicates that the debugging session is running APF authorized (supervisor state and system key).

When absent, the session is running in problem state and key.

Note, it is possible for Z/XDC to be running authorized while the program being debugged is not. When this is the case, this **(AUTH)** signal will still be displayed.

But normally, when debugging a non-authorized program, Z/XDC also will be non-authorized. If you need to change than, Z/XDC provides for a two-step process to do so. (The process takes two steps because it requires the cooperation of a separately running authorized program to initiate the process.) For more information, see **HELP COMMANDS SET AUTH.**

... **(DBCnnnW)** ---
When this is displayed, a warning condition exists. Use the **HELP MESSAGES DBCnnn** command the access more information.

Help FULlscreen TItle Right

Every fullscreen display that Z/XDC emits has a top line screen title that conveys generally useful information about the debugging session. There are three sections in the title line: a left side, a right side and a center section. Here, I will describe the **right side** information.

The title's right side shows either a scroll area position or the name of the job being debugged or any of several transient conditions pertaining to a command you've just issued. The following are possible:

nn,nnn
The default display shows your current scroll position either within a Built-in Help topic or within the Primary Window's scrollable history.

Note, if you ever wanted to know how many messages a given command emitted, just issue a second command (anything really) and compute the difference...

- Doing this for **LIST PGMS PLPA** will show you about how many load modules are contained in the System's Pageable Link Pack Area. (Hint, it's a LOT!)
- Issuing **M FFFFFF;L LK FFFFFF** will show you about how many csects are in the System Nucleus. (It's even more!)

A.S jobname
If you're connected (via CDF) to debugging session running in another address space, this shows the name of that address space.

n TASKS WAITING
If you're debugging a multitasking program, and multiple tasks have reached breakpoints (or ABENDed), then one task has gained control of the debugging session,

and the others (n of them) are awaiting their turn.

TOP-OF-DATA
BOTTOM-OF-DATA

You have issued a scrolling command **(UP, DOWN** or **LOCATE)** that has attempted to scroll past the top or bottom of the scroll area.

STRING FOUND
STRING NOT FOUND

You have issued a **SCANLOG** command to search the scroll area for a given string, and that string has either been found or not found.

PF KEY NOT DEFINED

You have pressed a function key whose meaning has not been defined.

MESSAGE NOT FOUND

You have issued a **LOCATE #nnn** command; however, the given command number (#nnn) is greater than the number of Z/XDC command strings that have been issued so far during the debugging session.

Y OR N REQUIRED

Z/XDC has displayed a query message **(DBC848Q** for example) that requires a yes/no answer before you are allowed to continue. You will need to respond as required.

ENTER REQUIRED
Z/XDC has displayed a message that it wants to make sure you see! You must acknowledge (by pressing the ENTER key) before you are allowed to continue.

STEP IN -> STEP OVER

You are using the **STEP IN** command to step through logic in a C program, and execution has reached a language statement that calls a program that Z/XDC is unable to allow you to step into. (Perhaps because it is not mappable?) This message alerts you to c/XDC’s action of processing the command as a **STEP OVER** instead.

MORE STORAGE APPENDED

You have a storage display in view, and you’ve pressed the **Page Down** key (or issued a **DOWN** command), but you’re already at the bottom of the window’s scroll area. Well rather than failing the command, Z/XDC, under the covers, has issued a **DISPLAY** or **FORMAT** command to append an additional storage display to the scroll area. It then positioned you into that newly appended display.

This message alerts you that that has happened.

Help FULlscreen Fsinputs

Linemode displays are very limited in the kinds of inputs they will accept from the terminal. Basically, only discrete commands can be typed. Fullscreen displays, on the other hand, are much more versatile. They can accept and act upon input information from many different kinds of fields from all over the displayed image, including traditional commands from one or more command lines. In particular, Z/XDC's fullscreen displays generally accept the following kinds of inputs:

COMMAND LINES
Z/XDC's fullscreen displays always include at least one (and often more) command lines where traditional Z/XDC commands can be typed. These command lines are pointed to by arrows **("===>").** When more than one command line is displayed, then they segment the terminal screen into multiple "windows" each of which can display information independently of the others.

The command line displayed near the top of the terminal screen is called the "primary" command line. This command line is not deletable, so it always exists. If additional command lines exist, then they are called "Watch Window" command lines. They can be created and removed at will via the **SET WINDOW CREATE/DELETE** commands (factory default **F1** and **shift-F1,** respectively).

Any Z/XDC command can be issued from the primary command line; however, only display type commands should be issued from Watch Window command lines. This is because the Watch Window command lines retain their command strings for automatic re-execution each time the ENTER key or any PF key is pressed. Accordingly, whenever you type a new command string, Z/XDC will execute it immediately, and then it will edit the string to remove from it all nondisplay type commands, and only what's left will be saved on the Watch Window command for subsequent automatic re-executions. See **HELP FULLSCREEN WINDOWS** for more information.

SHORTCUTS
Most lines displayed at the terminal have underscores (_) or periods (.) showing at the left side of the screen. These are shortcut input fields, and any of several 1- and 2-character shortcuts can be entered there. Different display lines accept different sets of shortcuts, but a **?** can always be typed to show the specific shortcuts accepted on a particular line. See **HELP SHORTCUTCMDS** for more information.

ASSOCIATED ADDRESSES
Most lines of information generated by Z/XDC are related to some object or location in storage. For example:

- In a **LIST TASKS** display, each line describing a TCB is related to the address of that TCB.
- Each line in a **LIST RBS** display is related to the location pointed to by the described RB's resume PSW.
- Each line in a **LIST PGMS** display is related to the described load module's entry address.
- Etc.

Accordingly, for those display lines that have related addresses, Z/XDC provides a Shortcut Entry Field where you can type display-type shortcuts such as **D F SH W L N** etc. See **HELP SHORTCUTCMDS ASSOCADDR** for more information.

AD-CONS

Many displays show hex data grouped into "words" that are 2, 3, or (most often) 4 bytes long. Often such words contain address constants ("ad-cons") that point to other locations in storage. Z/XDC permits you to display such referenced locations simply by typing any of several possible 1 or 2 character "Point-and-Shoot commands" directly on top of a displayed ad-con. See **HELP PASCMDS** for more information.

S-CONS

When formatted machine instructions are displayed, the hex equivalents of the displayed instructions are shown as one or more halfword fields at the left. Those halfword fields that show a base register with a displacement are called "s-cons". As in the case with ad-cons, Z/XDC permits you to display the locations pointed to by such s-cons just by overtyping them with a "Point-and-Shoot command". Again, see **HELP PASCMDS** for more information.

BUILT-IN HELP CROSSLINKS

The information produced in response to **HELP** commands often make references to information contained in other **HELP** command displays. Consequently, you will notice that **HELP** topic paragraphs, such as this one, often have a Shortcut Entry Field at the left. These Entry Fields are **crosslinks** to related topics. The referenced information can be displayed simply by typing an **H** or an **S** into such a Shortcut Entry Field and then pressing the ENTER key. For more information, please see **HELP ABOUTHELP.**

Help FULlscreen Displayerrs

Z/XDC’s Fullscreen Support supports a wide variety of VTAM terminals, and it uses a wide range of control command sequences to do so. Sometimes (either because of a fault in the Fullscreen Support or because of an incorrect terminal definition in VTAM) Z/XDC can cause "screen erasure" errors to occur at the terminal.

When this happens, just press the **PA2, PA1,** or **ATTN** key five times. This causes VTAM to return five "reshow" requests in a row back to its caller. When Z/XDC detects this, it assumes that something has gone wrong, and so it automatically shuts down its fullscreen displays and switches to using linemode TPUTs.

You may then proceed with debugging via the linemode interface, or you may retry the fullscreen interface by typing the **SET TFS ON** command.

If the problem repeats itself, you may try issuing some of the following commands to circumvent it.

- Try issuing the **SET PRIMARYSIZE** or **SET SECONDARYSIZE** commands. For terminals that have multiple screen sizes, these commands affect the control sequences used by Z/XDC.

- Try issuing the **SET COLORS DDDD** command to restore default colors.

- Try issuing the **SET HILIGHT DDDD** command to restore default hilighting settings.

- If ISPF is running, then try issuing the **SET ISPF** command to resume fullscreen mode with ISPF's Display Services (instead of fullscreen TPUT/TGETs).

- On the other hand, if it's Z/XDC's ISPF display interface that is giving problems, try issuing the **SET TPUT** command to resume fullscreen mode with Z/XDC's fullscreen TPUT/TGETs.

If you determine that terminal error problems are not the fault of incorrect definitions in VTAM, then please contact us for assistance. See **HELP SUPPORT CONTACTUS** for information about how to do so.

Help FULlscreen Ispf

Z/XDC's fullscreen support is fully compatible with ISPF. If ISPF is available, then Z/XDC may make use of its Display Services for sending data to your terminal. Otherwise, Z/XDC provides most ISPF like services on its own.

In particular, Z/XDC provides profile dataset management services regardless of whether or not ISPF is available. This permits debugging session characteristics (window definitions, function key definitions, etc.) to be saved from session to session. Furthermore, the saved profile data is wholly compatible with ISPF, thus profile data can, for example, be saved when Z/XDC is running outside of ISPF, and then it can be used later when Z/XDC is running from within ISPF.

Function key definitions can be created and saved without regard to whether Z/XDC is running within or outside of ISPF either now or later.

When ISPF is available, Z/XDC supports the use of the SPLIT and SWAP commands to permit the user to jump back and forth between a Z/XDC debugging session and another program that might or might not be related to the debugging work.

Whether or not Z/XDC uses ISPF's Display Services is controlled by several factors:

- Z/XDC must be running within a subtask of ISPF. If ISPF is not running in TSO, or if ISPF is running but Z/XDC is running within a task that is not a descendant of ISPF, then Z/XDC cannot use ISPF's Display Services. This is an ISPF restriction.

- If Z/XDC is being used to debug an authorized program, then Z/XDC cannot use ISPF's Display Services even when Z/XDC is started from within ISPF. This also is an ISPF restriction. In fact, if you do start an authorized debugging session from, say, ISPF option =6 or from Z/XDC's Startup Panel in ISPF, and then if you issue Z/XDC's LIST TASKS command, you will immediately see that ISPF has caused the debugging session to be started within a subtask that is outside of ISPF's task structure.

- If Z/XDC is running non-authorized within a subtask of ISPF, then Z/XDC's **SET ISPF** command can be used to turn on Z/XDC's usage of ISPF's Display Services. You then will be able to use ISPF's **SPLIT** and **SWAP** commands to switch back and

forth between the debugging session and other programs running elsewhere within ISPF.

You may notice that the terminal screen's title line changes from "Z/XDC **TPUT** INTERFACE" to "Z/XDC **ISPF** INTERFACE". See **HELP FULLSCREEN TITLE CENTER** for more information.

- You can use the **SET TPUT** command to cause Z/XDC to discontinue using ISPF's Display Services. Z/XDC will then start using its own fullscreen TPUTs and TGETs to write and read the terminal. Also, you will notice that the screen's title line changes back from "Z/XDC **ISPF** INTERFACE" to "Z/XDC **TPUT** INTERFACE". See **HELP COMMANDS SET TPUT** for more information.

The SET ISPF/TPUT switch can be saved in your session profile. To do so, type the following commands:

PROFILE READ
This command merely restores all profiled settings to your personal default values. That will prevent you from unexpectedly changing extraneous profile values.

SET ISPF [for example]
This command causes Z/XDC to use ISPF Display Services whenever it can. (When Z/XDC can't, it falls back to using its own fullscreen TPUTs and TGETs.)

PROFILE SAVE
This command saves your own personal copy of your new profile settings.

For more information about session profiles, see **HELP PROFILES.**

Help FULlscreen Logging

The Scroll Area

Whenever any command is issued (including storage display commands), all of the messages generated by that command are appended to the bottom of a Scroll Area, and then the display window is positioned to show at least the starting portion of those messages.

For the Primary Window, all prior commands and displays are retained (up to 10,000 lines worth), and they can be reached by simple scrolling commands. The beauty of this is that it makes your debugging session's history instantly accessible!

For more information about Scroll Areas, see HELP FULLSCREEN SCROLLING.

The Session Log

In addition, all messages that are displayed in the debugging session's **Primary Window** are also written to an external log file. This log may be a sequential DASD dataset but it is most convenient if it is a SYSOUT file on spool.

Logging may be switched dynamically from one dataset or spool file to another during a debugging session, but there is rarely any reason to do this.

The Session Log records all activity that takes place in Z/XDC's **Primary Window.** This includes all commands issued from that Window and all displays, reports, and actions generated by those commands.

Z/XDC's factory default settings for logging cause it:
- Always to be created.
- To be written to spool in SYSOUT class **X.**
- To be written with the **HOLD** attribute set. (See JES2 documentation if you want to know what that means.)

These are the best choices for several reasons:

- Although you will need to have a log only occasionally, if it is always created, then you will not have to predict when it is needed, and it will always be available when you do need it.

- On the other hand, if a log is always created, that creates a proliferation problem: Somebody/something is going to have to cleanup/delete old unneeded log datasets.

- But by writing the logs to a HOLD class on spool, the cleanup problem is solved because most data centers have JES2 settings that cause hold class files to be purged from spools once they become more that a few days old.

Latent Commands

Activity in the Watch Windows is **not** recorded, but typically a Watch Window is where PSW and register displays are presented, and those things in particular would be really really useful to record in the Session Log.

To address this need, Z/XDC provides something called **Latent Command Strings.** These are commands:
- That are automatically issued whenever execution passes from the user program to Z/XDC (such as when tracing through execution),
- But their outputs are written **only** to the **Session Log,**
- While their outputs are **normally hidden** in the **Scroll Area.**
- (But that can be overridden.)

For Assembler debugging, the default Latent Commands String is **L PSWE;L BEA;L RWREGS,** but that can be overridden too.

More Information

For more information, please see:
- HELP COMMANDS LOG
- HELP COMMANDS SET LOG
- HELP COMMANDS LIST LOG
- HELP FULLSCREEN LOGGING LATENTCOMMANDS

Help FULlscreen Logging Latentcommands

Latent Commands are commands:
- That are automatically executed whenever execution passes from the user's program into Z/XDC (via breakpoints usually),
- That normally are not recorded in the Scroll Area,
- But that are recorded in the Session Log.

Why

The purpose of Latent Commands is to provide a reliable means of recording critical information into the Session Log. Typically that would include the Registers and PSW.

Normally, such information is presented to the user in a Watch Window, but Watch Window displays are not written to the Session Log. So without Latent Commands, critical information would be lost.

When

Latent Commands Strings are processed whenever Z/XDC receives control from the user's program. Typically, this would be:
- Either because execution reached a trap of some sort,
- Or because the user was stepping execution through his code.
- But it could also be because the user's program has ABENDed.

What

The default Latent Commands String is **L PSWE;L BEA;L RWREGS.**

But this can be overridden with the **SET LOG LATENT='whateveryouwant'.**

Also, whatever you choose as your favorite Latent Commands String can be saved into your Session Profile.

Where

The Latent Commands and the reports they produce are always written to the Session Log.

They normally are not written into the Scroll Area. This is because the information typically is already visible in a Watch Window, and making it visible in the Scroll Area would be visually disruptive, especially to **UP TRACE** commands.

However, if you want to see them in the Scroll Area anyway, you can enable that with

a **SET LOG LATENT=SHOW** command, and you can save that setting into your Session Profile as well.

More information

The **Factory Default** Latent Commands String is **L PSWE;L BEA;L RWREGS.**

Your personal default Latent Commands String comes from your Session Profile:
- It can be displayed and set from the **Profile Menuing System.**
- It also can be set by the **SET LOG LATENT=** command.
- It also can be displayed by the **LIST LOG** command.

Help FULlscreen Notes

Z/XDC provides an "annotation" facility for creating, displaying, and maintaining a list of addresses of interest and notes and comments about those addresses. This is called the **Notes List.**

There are two ways to add an annotated address to the **Notes List:**
- The **NOTE** command can be used to:
  - Specify an address of interest,
  - Assign an initial comment to that address,
  - And to add that address, along with the comment, to the **Notes List.**
- The **N** ("Note") shortcut can be used on any display line having an associated address (such as a storage display line). The associated address is used as the note entry's address, and the display line's text is used as the note's initial comment.

The currently accumulated list of annotated addresses can be displayed by the **LIST NOTES** command.

When a **Notes List** is displayed, any annotating comment can be changed simply by overtyping the displayed text.

Any **Notes List** entry can be purged from the list by typing a **P** shortcut in that entry's Shortcut Entry Field (shown by an underscore [_]).

When a note is purged from the list...
- It will not be displayed by subsequent **LIST NOTES** commands,
- But its entry in any previously issued **LIST NOTES** displays will still be retained in the debugging session's scrollable area,
- But that entry will immediately become **nonoverwritable.**

Help FULlscreen PFkeys

Like ISPF, Z/XDC allows you to define command strings for the various Program

Function Keys ("PF keys") on your terminal. However, Z/XDC provides several special features to make the management and use of PF keys more flexible and more powerful.

- PF key command strings may contain a hyphen (-) to define an insertion point for text, if any, from the command line.

- PF key commands may also include an underscore (_) to cause the command to pause for user input.

- PF key definitions are grouped together by Z/XDC into "PF key sets" of 12, so that you can more easily assign your "preferred" PF key definitions to those function keys that are "easiest" to use at your terminal.

- Z/XDC supports the creation of up to 36 PF key definitions (3 PF key sets) for use at terminals that can send that many function key codes.

- Z/XDC permits the creation of a separate set of PF key definitions for use only during Built-in Help Displays.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

HYPHEN - Using the hyphen (a.k.a. "dash" or "minus") in PF key definitions.
UNDERSCORE - Using the underscore character (_) in PF key definitions.
PFKEYSETS - Assigning sets of PF key definitions to sets of function keys at your terminal.
DEFAULTKEYS - The factory default PF key definitions and what they mean.
HELPKEYS - PF key definitions used during Built-in Help.
TRACING - PF key definitions used for program tracing. (See **HELP BREAKPOINTS TRACING PFKEYS.)**

Help FULlscreen PFkeys HYphen

PF key command strings may contain a single hyphen (-) to identify an insertion point for text from the command line. When you use the PF key while the cursor is within the terminal screen's Primary Window, and if the Primary Window's command line contains text, then the first single hyphen occurring in the PF key definition is replaced by the text from the command line.

If a PF key does not contain any single hyphens, **or** if the PF key is issued when the cursor is within a Watch Window, then the command line text, if any, is ignored.

If you want to prevent this substitution from occurring, then double up the hyphen. If Z/XDC encounters a doubled hyphen before encountering a single hyphen, then when the PF key is issued, Z/XDC will issued the PF key's command with the doubled hyphen replaced by a single hyphen.

If Z/XDC encounters single or doubled hyphens **after** the first single hyphen, then Z/XDC treats them as normal text to be processed without substitution or replacement.

Note that "hyphens", "dashes", and "minus signs" are all the same character.

Example:

PF key DEF	**CMDLINE TXT**	**RESULT**
F +0	JUNK	F +0
F -0	JUNK	F JUNK0
F --0	JUNK	F -0
EQ X A--B+-+5--3	C	EQ X A-B+C+5--3
EQ X A--B+-+5-3		EQ X A-B++5-3
DOWN -	5	DOWN 5
DOWN	5	DOWN
UP -		UP
TSO -	FREE ALL	TSO FREE ALL

Help FULlscreen PFkeys Underscore

PF key command strings may contain a single underscore (_) to cause Z/XDC to pause for user input before issuing the command.

When a function key is pressed, if the assigned command string contains a single underscore, Z/XDC will remove the underscore from the command string, display the command string on the command line, place the cursor at the point where the underscore had been, and unlock the keyboard so that you can modify the command string before pressing ENTER.

If you want to prevent this substitution from occurring, then double up the underscore. If Z/XDC encounters a doubled underscore before encountering a single underscore, then when the PF key is issued, Z/XDC will issued the PF key’s command with the doubled underscore replaced by a single underscore.

If Z/XDC encounters single or doubled underscores **after** the first single underscore, then Z/XDC treats them as normal text to be processed without substitution or replacement.

Example:

PF key DEF	**RESULT**	**CURSOR PLACEMENT**
F +_	F +	following the "+"
EQ X__Y PSW?--_ FLOAT	EQ X_Y PSW?- FLOAT	following the "-"

Help FULlscreen PFkeys Pfkeysets

This Topic is Largely Obsolete

The problems addressed here no longer occur. Nevertheless, once upon a time there were considerable inconsistencies in the number of and layout of PF keys from one terminal to the next. That no longer is the case, but the support for this still exists within Z/XDC, so...

THE PROBLEM:

The keyboards for mainframe terminals can have many many different layouts. In particular, the function keys can be presented on different keyboards in a variety of ways. Some terminal have 24 function keys presented in two rows of 12 each running across the top of the keyboard. Other terminals use just one row (or cluster) of 12 function keys. In this second case, the keys are actually "multi-function" keys. For example, function key number 1 might be used for signaling either F1 or shift-F1. The user would be required to hold down SHIFT or CTRL to cause the key to send the second code.

Still other terminals have just 10 function keys and, therefore, are not able to transmit the normal full set of 24 PF key codes. These terminals might be able to transmit only PF key codes 1-10 and 11-20, **or** they might instead be able to transmit only PF key codes 1-10 and 13-22!

Still other terminals might have no separate function keys at all, but instead require the user to hold down an ALT key while pressing an appropriate numeric or alphabetic key.

Finally, some terminals have ways to transmit up to 36 different PF key codes!

If a terminal uses multi-function keys, then one set of PF key codes is going to be "easier" to type than the other. For example, if a terminal has a 12-key cluster labeled PF13 through PF24, then in order to transmit PF key codes 1-12, you will have to hold down SHIFT (or CTRL or FUNCTION) while pressing the appropriate function key. In this case, PF key codes 13-24 are easier to type than codes 1-12. With other terminals, the **exact opposite** will be true!

Probably, you would want your "preferred" PF key commands to be assigned to the set of function keys that are easiest to type, and you would like your secondary PF key commands to be relegated to the less convenient function keys.

The problem is that for some terminals (particularly the older IBM manufactured terminals) the easiest function keys are numbers 13-24, but for other terminals (such as PC terminals with "101 key" keyboards) the exact opposite is true: function key numbers 1-12 are easier to type.

THE SOLUTION:

In order to solve some of the problems presented by these various circumstances, Z/XDC groups your PF key definitions into up to 3 **PF key sets** (labeled **SET-A,**

SET-B and **SET-C)** each containing 10 or 12 definitions. Z/XDC then lets you assign each PF key set to a range of function keys available at your terminal. For example, you would probably create your preferred PF key definitions in **SET-A** and your secondary definitions in **SET-B.** Then if your easiest to use function keys were 13-24, you would assign **SET-A** to that key range and **SET-B** to the 1-12 key range. On the other hand, if your easiest keys were 1-12, then you would do the opposite.

If your terminal supports 36 function keys, then you can create a third set of PF key definitions in **SET-C** and assign that set to the 25-36 key range.

If your terminal has only 10 function keys (such as PC terminals with older keyboards), then it probably can transmit only 20 PF key codes. The "easier" PF key codes would probably be 1-10. Depending upon your workstation software, the secondary codes could be either 11-20 **or** 13-22! To handle this situation, Z/XDC allows you to assign a PF key set to each key range. In each PF key set, only the first 10 definitions will be active, the 11th and 12th definitions will be inactive. The key set assigned to the secondary key range can be assigned to start either at F11 or shift-F1, whichever is appropriate for you terminal's workstation software.

To create PF key definitions, use Z/XDC's **KEYS** command.

To create PF key definitions for use during Built-in Help displays, use Z/XDC's **HKEYS** command.

To assign PF key sets to terminal function key ranges, you need to bring up the "PFKEYS SETS TO RANGES ASSIGNMENTS" panel in the Profile Menuing System. This panel can be reached in any of several ways. One way is:

- Enter the Profile Menuing System by typing the **PROFILE** command without operands.
- From the Profile Menuing System's initial panel, select "Display/Change Debugging Session PF keys.".
- From the PF key display panel, select "Select here to view and update the PF key sets to ranges assignments.".

Another way is:

- Enter the PF keys Display/Update menu by typing the **KEYS** command (or by selecting that menu from the Profile Menuing System's initial menu).
- From the PF key display panel, select "Select here to view and update the PF key sets to ranges assignments.".

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

SUGGESTIONS - Suggestions and examples of assigning PF key sets to function key ranges for specific terminals.

Help FULlscreen PFkeys Pfkeysets Suggestions

This topic is largely obsolete. The problems addressed here no longer occur. Nevertheless, once upon a time there were considerable inconsistencies in the number of and layout of PF keys from one terminal to the next. That no longer is the case, but the support for this still exists within Z/XDC, so...

Z/XDC supports the creation of three sets of PF key definitions. These sets are named SET-A, SET-B, and SET-C. Each set holds 12 PF key definitions. Each set can be assigned to one (or more) ranges of function keys at your terminal. Example: If you assign SET-C to function key range 13-24, then when you press function key 14, Z/XDC will execute the second PF key definition in SET-C.

Programmatically speaking, the three PF key sets are equivalent. For consistency, however, I would like to suggest that the three sets be used as follows:

SET-A - This should contain the PF key definitions that you expect to use most frequently. You should assign this set to your terminal's function key range that is the easiest for your fingers to reach.

SET-B - This should contain the PF key definitions that you expect to use less frequently. You should assign this set to your terminal's function key range that is less easy for your fingers to reach.

SET-C - This should be used only if you are using a terminal that can transmit 36 function key codes. This set should be assigned to function key range 25-36.

Z/XDC supports the assignment of PF key sets to the following terminal function key ranges:

01-12 - Function keys 1 through 12 are available on all terminals except PCs using older keyboards having only 10 function keys. On certain older IBM 3270, 3279, and other terminals, function codes 1 through 12 are sent by shifting function keys 13-24.

13-24 - Function keys 13 through 24 also are available on all terminals except PCs using older keyboards having only 10 function keys. They also are not available on certain very old IBM terminals having only keys 1-12.

25-36 - A few terminals are capable of issuing up to 36 function key codes. For example, IBM's 7171 Protocol Converter supports 36 function keys for connected ASCII terminals.

01-10 - If your terminal is a PC with an older keyboard having only 10 function keys, then you should assign your SET-A PF key definitions to this range.

11-20 - If your terminal is a PC with an older keyboard having only 10 function keys, then the shifted F1 key may transmit either function code 11 or 13, depending upon the workstation software that you are using. If shift-F1 sends function code 11, then you should assign your SET-B PF key definitions to this range.

13-22 - On the other hand, if your workstation software causes shift-F1 to send function code 13, then you should assign your SET-B PF key definitions to this range.

The following are suggested PF key set assignments for specific terminals.

IBM 327x TERMINALS
Not to be confused with PC terminals running workstation software that is only "emulating" 327x terminals. I'm talking about actual IBM 3272, 3279, and other 327x terminals or their brand-x equivalents. These older IBM terminals have a cluster of function keys labeled PF13 through PF24 located at the right-hand side of the keyboard. For these terminals, you should assign SET-A to the 13-24 key range and SET-B to the 01-12 key range. The **SET KEYS ASIS** command can be used to establish this arrangement.

PC TERMINALS WITH "101 STYLE" KEYBOARDS
These terminals have a single row of function keys labeled F1 through F12. For these terminals, you should assign SET-A to the 01-12 key range and SET-B to the 13-24 key range. The **SET KEYS SWAP** command can be used to establish this arrangement. Also, this is Z/XDC's factory default arrangement; therefore, this arrangement can be reestablished by both the **PROFILE RESET** command and the **SET KEYS RESET** command.

IBM 3179 TERMINALS
These IBM terminals have two 12-key rows of function keys running across the top of the keyboard. For these terminals, you should assign SET-A to whichever row you find easier or more convenient to type. You should assign SET-B to the other row.

PC TERMINALS WITH ONLY 10 FUNCTION KEYS
You should assign SET-A to the 01-10 key range. You should then assign SET-B either to the 11-20 or the 13-22 key range depending upon the workstation software that you are running. If your software transmits shift-F1 as function key 11, then you should use the 11-20 range. On the other hand, if your software transmits shift-F1 as function key 13, then you should use the 13-22 range.

PROTOCOL CONVERTER CONNECTED TERMINALS
Protocol converters allow ASCII terminals and PCs running communications software such as PROCOMM, CROSS-TALK, or KERMIT to connect to IBM mainframes by dial-up connections using asynchronous modems. A protocol converter converts the communication protocols back and forth between asynchronous ASCII and synchronous 327x. IBM's 7171 Protocol Converter (and perhaps others) can permit connected terminals to send up to 36 function key codes. If you are using one of these terminals, then you should assign PF key SET-C to the 25-36 function key range.

Help FULlscreen PFkeys Defaultkeys

Your personal PF key settings are a part of your personal profile settings. To load (or restore) your personal settings (including your favorite PF key settings), use the **PROFILE READ [name]** command.

To load your Data Center's default PF key settings (as well as all other profiled settings), use the **PROFILE READ XDC TFS** command.

To load Z/XDC's factory default PF keys and other settings, use the **PROFILE RESET [name]** command.

Z/XDC organizes its PF key definitions into sets of 12 each. These sets can than be easily assigned to available ranges of function keys at your terminal. See **HELP FULLSCREEN PFKEYS PFKEYSETS** for more information.

The default PF key definitions are grouped into three PF key sets: **SET-A, SET-B,** and **SET-C.**
- The default Sets A and B are similar but have some differences.
- Default Sets B and C are identical.

The definitions in default SET-A are "preferred". I think that generally you will be using these PF keys more frequently than those defined in SET-B. Accordingly, you should be sure to assign the SET-A definitions to those function keys at your terminal that are "easier to use". For all present day terminals, that would be function keys 01-12. (For more information, see **HELP FULLSCREEN PFKEYS PFKEYSETS.)**

Subtopics Index

For more information, select the following topics. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

SET-A - Descriptions of the factory default PF key definitions in SET-A.
SET-B - Descriptions of the factory default PF key definitions in SET-B and SET-C.

Help FULlscreen PFkeys Defaultkeys SET-A

SET-A contains the **preferred** PF key definitions. This set should be assigned to your terminal's function keys that are "easier" to use. By default, this set is assigned to PF keys **F1** thru **F12.**

There are **four** Factory Default profiles. The factory default settings for PF key SET-A are mostly the same across all of the default profiles. However, the settings do differ for the 9th, 10th and 11th keys in SET-A. Those differences are shown below. For more information about Factory Default profiles, see **HELP PROFILES FACTORYDEFAULTS.**

1st - (F1) **SET WINDOW DELETE** (All Factory Default profiles)
This PF key removes a Watch Window command line from your terminal screen. The window to be deleted is implied by the location of the cursor. Simply move the cursor into the target window (anywhere will do) and press F1. The window containing the cursor will disappear. (The corresponding PF key definition in SET-B is **SET WINDOW CREATE.)**

2nd - (F2) **SPLIT** (All Factory Default profiles)
This PF key is identical to ISPF's **SPLIT** command. It creates (or switches to)

a 2nd ISPF window in which you can do work in ISPF, not related to Z/XDC, without having to terminate Z/XDC. If no 2nd window exists, then one is created with the split line located at the position of the cursor when this PF key was pressed. On the other hand, if a 2nd ISPF window already exists, then focus is switched to it with the split line moved to the cursor location.

To use this key, first move the cursor to the location on the screen where you want the ISPF **split line** to appear (or be moved to if a split line already exists). Then press this key.

Important: This **SPLIT** key can be used only when **all** of the following are true:

- Z/XDC is running non-authorized.
- Z/XDC is running under ISPF.
- The **SET ISPF** command is in effect.
- **Or** you are debugging (via cdf/XDC) a background job (authorized or not) from a TSO terminal session.

Once you have used this command to create a second ISPF window, you can then use **SET-B's** 9th PF key **("SWAP -")** to switch control back and forth between the ISPF work and the Z/XDC debugging session. (The corresponding PF key in SET-B is **SPLIT NEW.)**

3rd - (F3) **END** (All Factory Default profiles)

This PF key can be used to terminate any of the following:

- A Built-in Help display.
- A menu in Z/XDC's Profile Menuing System.
- A **LOGON** panel or **Job Selection Menu** in cdf/XDC.
- Z/XDC's **License Control Data Alter/Display** panel.

This key can also be used to cause Z/XDC to percolate the current ABEND to the next older ESTAE or ESTAI recovery routine. Usually, this will end a debugging session, but not necessarily. If Z/XDC is present in multiple ESTAE/ESTAI routines, then Z/XDC will be entered again, and debugging can continue under the control of the older ESTAE/ESTAI routine. Also, because the **END** command does irrevocably percolate the debugging session, a "second chance" prompt is displayed asking if you really do mean to issue the **END** command. You can then confirm or abort your decision. (The corresponding PF key in SET-B and in the Built-in Help PF key set also is **END.)**

4th - (F4) **END COMPLETELY** (All Factory Default profiles)

This PF key can be used as follows:

- If issued from Z/XDC's normal debugging session display, then it absolutely terminates the debugging session. The current ABEND is percolated to the next older ESTAE or ESTAI recovery routine. If Z/XDC is present in any older recovery routines, that it automatically continues the percolation. Thus if you really do want to terminate a debugging session, using this 4th PF key is better than the 3rd because (a) no "second chance" prompt occurs, and (b) any reentries into Z/XDC that may occur from older ESTAE/ESTAI routines are automatically terminated.
- If issued from a menu in Z/XDC's Profile Menuing System, this key

functions like the **QUIT** command: The Menuing System is immediately exited and all profile changes are kept.

- If issued from Z/XDC's **License Control Data Alter/Display** panel, this key functions like the **CANCEL** command: The panel is immediately exited and all changes are discarded.

- This key cannot be issued from a Built-in Help display because the 4th PF key definition for Built-in Help is entirely different.

(The corresponding PF key in SET-B, but **not** in Built-in Help, also is **END COMPLETELY.)**

5th - (F5) **FIND** (All Factory Default profiles)
This PF key can be used to repeat or resume a search of virtual storage for a previously given character or hex string. It **cannot** be used as follows:

- It cannot be used to search for a new string. You must instead type the **FIND** command along with its operands and then press the ENTER key.

- It cannot be used to search your session log. Use the **SCANLOG** command instead.

(The corresponding PF key in SET-B is **SCANLOG -.)**

6th - (F6) **TSO -** (All Factory Default profiles)
This PF key can be used to issue a native TSO command (ALLOC, FREE, LISTD, etc.). To use this key, first place the desired TSO command on the command line, then press this key. The specified command will be executed as a subtask of the task under which Z/XDC is currently running. (The corresponding PF key in SET-B also is **TSO -.)**

7th - (F7) **UP -** (All Factory Default profiles)
This PF key is similar to ISPF's **UP** key. It causes the display of your debugging session log to be scrolled upwards by a specified or default amount. You may type an operand on the command line, before pressing this key, that allows you to specify the upwards scroll distance.
Z/XDC's **UP** command accepts the same operands as ISPF's **UP** command. (The corresponding PF key in Built-in Help also is **UP -.** The corresponding PF key in SET-B is **UP M.)**

8th - (F8) **DOWN -** (All Factory Default profiles)
This PF key is similar to ISPF's **DOWN** key. It causes the display of your debugging session log to be scrolled downwards by a specified or default amount. You may type an operand on the command line, before pressing this key, that allows you to specify the downwards scroll distance.
Z/XDC's **DOWN** command accepts the same operands as ISPF's **DOWN** command. (The corresponding PF key in Built-in Help also is **DOWN -.** The corresponding PF key in SET-B is **DOWN M.)**

9th - (F9) **T** (Factory Default profiles **A80** and **AWIDE** only)
This is suitable for Assembler programmers. It is one of three PF keys that

assist in the tracing of user program execution. This allows you to step through the execution of your program exactly one instruction at a time. (The corresponding PF key in SET-B is **SWAP NEXT.)**

9th - (F9) **STEP IN** (Factory Default profiles **C80** and **CWIDE** only)
This is suitable for C programmers. It steps execution from one language statement to the next. If the current statement calls a subroutine or invokes a function, **STEP IN** follows execution into the subroutine or function. (The corresponding PF key in SET-B is **SWAP NEXT.)**

10th - (F10) **T BY** (Factory Default profiles **A80** and **AWIDE** only)
This is suitable for Assembler programmers. It is one of three PF keys that assist in the tracing of user program execution. This allows you to step through the execution of your program from one successful branch type instruction to the next. (The corresponding PF key in SET-B **LEFT -.)**

10th - (F10) **STEP OUT** (Factory Default profiles **C80** and **CWIDE** only)
This is suitable for C programmers. If execution is within a subroutine or function, it allows that subroutine or function to run to completion, and it recaptures execution at the statement following the one that called the subroutine or invoked the function. (The corresponding PF key in SET-B **LEFT -.)**

11th - (F11) **T NXSEQ() NXSEQ(2) NXSEQ(3) NXSEQ(4) NXSEQ(5) NXSEQ(6);GOT** (Factory Default profiles **A80** and **AWIDE** only)

This is suitable for Assembler programmers. It can be used to "step over" subroutines; i.e., it can be used to bypass the tracing of subroutines.

If during **T** or **T BY** tracing, execution reaches a "branch and link" type instruction (see "BAL-type" instructions in **HELP BREAKPOINTS TRACING BRANCHES),** you have two choices of what to do next:

(a) - You may want to continue tracing execution into the subroutine, in which case, just continue to use the **T** or **T BY** (and other tracing commands) to proceed normally.

(b) - You may want to let the subroutine execute without being traced, but you want to recapture control of execution when the subroutine terminates. This can be accomplished by using this PF key.

This PF key also provides one way in which to bypass the tracing of repeated iterations of a loop. If you have reached the bottom instruction of a loop, and if you **know** that the loop eventually will exit to the next instruction following its bottom instruction, then use this PF key. **Warning:** if the loop should unexpectedly exit by branching to some other instruction, then you will lose control of the trace unless you take the precaution of first placing traps at all possible exit points. For a really good example, see **HELP PASCMDS INSTRUCTIONS T-EXAMPLE.**

This PF key definition is as complex as it is because it is designed to be usable for bypassing even subroutines that make vectored returns to their callers. Generally, such subroutines will return into a list of instructions immediately following the linking instruction.

However, this PF key can still be used for subroutines that make only normal returns. This is because the breakpoints set by this PF key are transient and will all be automatically removed by Z/XDC when the subroutine returns to the instruction just past its linking instruction.

(The corresponding PF key in SET-B is **RIGHT -.**

11th - (F11) **STEP OVER** (Factory Default profiles **C80** and **CWIDE** only)
This is suitable for C programmers. It steps execution from one language statement to the next. If the current statement calls a subroutine or invokes a function, **STEP OVER** allows that subroutine or function to run to completion and recaptures execution upon its return. (The corresponding PF key in SET-B is **RIGHT -.)**

12th - (F12) **RETRIEVE** (All Factory Default profiles)
This PF key can be used to recall the last previously issue command to the command line. If you press this key repeatedly, you will recall successively older commands until you exhaust the saved commands stack whereupon a warning message will be displayed. If you then continue to press this key, the retrieval process will start over with the newest command. (The corresponding PF key in SET-B is **RETRIEVE +1.)**

Help FULlscreen PFkeys Defaultkeys SET-B

SET-B contains a **secondary** set of twelve PF key definitions. This set should be assigned to those function keys at your terminal that are not as convenient to press as SET-A. By default, this set is assigned to PF keys **shift-F1** thru **shift-F12** (PF13-PF24). For more information, see **HELP FULLSCREEN PFKEYS PFKEYSETS.**

SET-C contains a set of PF key definitions that can be used only if you are lucky enough to have a terminal that supports **36** PF key definitions. However, it’s been over 30 years since I’ve seen one, so you can pretty much ignore SET-C definitions.

Anyway, Z/XDC’s factory default definitions for SET-C are identical to the definitions for SET-B, so SET-C will not be further discussed.

There are **four** Factory Default profiles. The factory default settings for PF key SET-B are identical across all of the default profiles. For more information about Factory Default profiles, see **HELP PROFILES FACTORYDEFAULTS.**

Many of the PF key definitions in SET-B are identical to the corresponding definitions in SET-A, and so they will not be described here either. See **HELP FULLSCREEN PFKEYS DEFAULTKEYS SET-A** for those descriptions. Those PF keys are:

3rd - (shift-F3) END
4th - (shift-F4) END COMPLETELY
6th - (shift-F6) TSO -

SET-B PF keys that Differ From SET-A

1st - (shift-F1) **SET WINDOW CREATE**
This PF key adds a Watch Window command line to your terminal screen. In effect a new Watch Window is created. Simply move the cursor to the desired line (any column will do) and press shift-F1. A new command line will appear at that screen line. (The corresponding PF key definition in SET-A is **SET WINDOW DELETE.)**

Alternatively, type into the command line the number of the line at which you want a new command line to appear, then press shift-F1. This will work too.

2nd - (shift-F2) **SPLIT NEW -**
This PF key is identical to ISPF's **SPLIT NEW** command. It creates a new ISPF window in which you can do work in ISPF, not related to Z/XDC, without having to terminate Z/XDC. (It never switches to an existing ISPF window.) It also will create a split line at (or move an existing split line to) the current cursor location.

To use this key, first move the cursor to the location on the screen where you want the ISPF **split line** to appear (or be moved to if a split line already exists). Then press this key.

Important: This **SPLIT NEW** key can be used only when **all** of the following are true:
- Z/XDC is running non-authorized.
- Z/XDC is running under ISPF.
- The **SET ISPF** command is in effect.
- **Or** you are debugging (via cdf/XDC) a background job (authorized or not) from a TSO terminal session.

Once you have used this command to create a new ISPF window, you can then use SET-B's 9th PF key **("SWAP -")** to switch control back and forth between the ISPF work and the Z/XDC debugging session. (The corresponding PF key in SET-A is just **SPLIT.)**

3rd - (shift-F3) See **F3** in **HELP FULLSCREEN PFKEYS DEFAULTKEYS SET-A.**

4th - (shift-F4) See **F4** in **HELP FULLSCREEN PFKEYS DEFAULTKEYS SET-A.**

5th - (shift-F5) **SCANLOG -**
This key is similar to ISPF's **FIND** key. It can be used to search your window's scroll area for a given character string.

To use this key, first type the desired search string on the command line, then press shift-F5. The window's scroll area will be searched until the string is found.

If no string or operand is given, then the previously issued search is repeated.

(The corresponding PF key in SET-A is **FIND** for searching **storage,** [not the window's scroll.])

6th - (shift-F6) See **F6** in **HELP FULLSCREEN PFKEYS DEFAULTKEYS SET-A.**

7th - (shift-F7) **UP M**
This a "scroll up max" PF key. It causes the display of your debugging session log to be scrolled upwards all the way to the top (oldest) entry in the log. (The corresponding PF key in both SET-A and Built-in Help is **UP -.)**

8th - (shift-F8) **DOWN M**
This a "scroll down max" PF key. It causes the display of your debugging session log to be scrolled downwards all the way to the bottom (newest) entry in the log. (The corresponding PF key in both SET-A and Built-in Help is **DOWN -.)**

9th - (shift-F9) **SWAP NEXT**
This PF key is similar to ISPF's **SWAP** command. When you have created multiple ISPF windows, it switches to the next in line so that you can do work in ISPF that is not related to Z/XDC. The ISPF window must have been already created via a **SPLIT** or **SPLIT NEW** command (the 2nd PF key).

Important: This **SWAP NEXT** key can be used only when **all** of the following are true:
- Z/XDC is running non-authorized.
- Z/XDC is running under ISPF.
- **Or** you are debugging (via cdf/XDC) a background job (authorized or not) from a TSO terminal session.
- The **SET ISPF** command is in effect.
- A **SPLIT** or **SPLIT NEW** command has previously been issued that created a 2nd (or 3rd or 4th ...) ISPF window that can be **SWAP'd** to.

(The corresponding PF key in SET-A is either **T** or **STEP IN,** depending upon which Factory Default profile has been loaded.)

10th - (shift-F10) **LEFT -**
This PF key is similar to ISPF's **LEFT** key. It causes the display of your debugging session log to be scrolled leftwards by a specified or default amount. You may type an operand on the command line, before pressing this key, that allows you to specify the leftwards scroll distance.
Z/XDC's **LEFT** command accepts the same operands as ISPF's **LEFT** command. (The corresponding PF key in SET-A is either **T BY** or **STEP OUT,** depending upon which Factory Default profile has been loaded.)

11th - (shift-F11) **RIGHT -**
This PF key is similar to ISPF's **RIGHT** key. It causes the display of your debugging session log to be scrolled rightwards by a specified or default amount. You may type an operand on the command line before pressing this key. That allows you to specify the rightwards scroll distance.
Z/XDC's **RIGHT** command accepts the same operands as ISPF's **RIGHT** command. (The corresponding PF key in SET-A is either **T SEQ() NXSEQ(2)...;GOT** or **STEP OVER,** depending upon which Factory Default profile has been loaded.)

12th - (shift-F12) **RETRIEVE +1**
The corresponding PF key definition in SET-A is **RETRIEVE** and is used to recall to the command line a previously issued command. If you make the mistake of pressing the **RETRIEVE** PF key (F1) too many times, you can then use this **RETRIEVE +1** PF key (shift-F1) to move forward in the command stack to get back to the command that you missed.

Help FULlscreen PFkeys HElpkeys

Whenever a Built-in Help display is sent to the terminal, the normal set of PF key definitions is replaced with a special set that is more suitable for navigating around the Built-in Help database. Normally, these **HELP PF key** definitions are displayed at the bottom of your screen both to remind you that special definitions are in effect and to show you what those definitions are. You can, however, turn this display on or off with the **SET HKEYS SHOW/NOSHOW** command.

For information about the organization of the Built-in Help database and about accessing the database, see **HELP ABOUTHELP.** The database is organized into an inverted tree structure, and many of the **HELP PF key** commands follow various pathways through that structure. In order to understand these PF key commands, you first need to understand the database structure as described in **HELP ABOUTHELP.**

The **HELP PF keys** contain 12 definitions. When the **HELP PF keys** are in effect this set of 12 definitions is assigned both to keys 1-12 and to keys 13-24 (and to keys 25-36 when present). Thus for example, when the **HELP PF keys** are in effect, PF keys F1 and shift-F1 (and PF25) issue the identical command.

Before I get into the specific **HELP PF key** descriptions, please take note of the following hints:

- In general, a Built-in Help topic will be longer than the available display space at your terminal. Therefore, you will have to use PF keys **F7** and **F8** to scroll **up** and **down** within a particular topic.

- A selected Built-in Help topic may continue across multiple topics. Therefore, you will have to use PF keys **F10** and **F11** to advance **backwards** and **forwards** through a topic.

- **F6** can be used to display an **index** for the Built-in Help database. This index is a powerful navigation tool.

- To **exit** Built-in Help, use **F3.** (Do **not** use F4.)

The factory default definitions for the **HELP PF keys** are as follows. (These definitions can be changed using Z/XDC's **HKEYS** command):

1st - (F1) **HELP *UP**

The current position in the Built-in Help database is move upwards along the current parent-child pathway. In other words, the current topic's "mother" topic is displayed.

2nd - (F2) **HELP *DOWN**

The current position in the Built-in Help database is move downwards along one of the possible downwards pathways. In other words, one of the current topic's "daughter" topics is displayed.

3rd - (F3) **END**

Built-in Help is terminated, and control is returned to the normal debugging session screen that was displayed prior to Built-in Help being entered. Also, the PF key definitions are restored to their normal non-HELP values.

4th - (F4) **HELP *BACK**

The current position in the Built-in Help database is move backwards along the current sister path.

5th - (F5) **HELP *FORWARD**

The current position in the Built-in Help database is move forwards along the current sister path.

6th - (F6) **LIST HELP *UP**

This displays a local index for the area of the Built-in Help database in which you are currently positioned. You can then use **L** and **H** shortcuts to quickly move to other areas of the database and to select another topic for display. Note the following warnings, however:

- Although it is not obvious, this command causes Z/XDC to exit Built-in Help. Therefore, when this command's display is in view, the **HELP PF key** definitions are **not** in effect.

  However, if you then use the **H** shortcut to select another topic for display, you will be returned to Built-in Help and the HELP PF key definitions will again be in effect.

- If the Built-in Help topic currently being displayed is the Master Index topic (i.e. the database's top topic). then this PF key will fail since there is no topic that is *UPwards of the top topic (just like there is nothing north of the north pole).

7th - (F7) **UP -**
This is a normal scrolling command. It shifts the display upwards within the Built-in Help topic currently being displayed.

8th - (F8) **DOWN -**
This is a normal scrolling command. It shifts the display downwards within the Built-in Help topic currently being displayed.

9th - (F9) **SWAP -**
This key is similar to ISPF's **SWAP** command. It switches to another ISPF window in which you can do work in ISPF that is not related to Z/XDC. The ISPF window must have been already created via a **SPLIT** or **SPLIT NEW** command.

This PF key accepts operands from the command line. For example, if you wanted to issue a **SWAP LIST** command, you could type **LIST** onto the command line, and then press this PF key. The command line contents would be substituted for the hyphen in the PF key's definition, and so the **SWAP -** command would become a **SWAP LIST** command.

If this PF key is pressed when the command line is empty, then the **SWAP -** commands becomes simply **SWAP.**

Important: This **SWAP -** key can be used only when **all** of the following are true:

- Z/XDC is running non-authorized.

- Z/XDC is running under ISPF.

- The **SET ISPF** command is in effect.

- A **SPLIT** or **SPLIT NEW** command has previously been issued that created a 2nd (or 3rd or 4th ...) ISPF window that can be SWAP'd to.

10th - (F10) **HELP *PREVIOUS**
The current position in the Built-in Help database is move backwards along the depth-first sequential pathway. In other words the current topic's next preceding topic is displayed. This may be a sister topic or the mother topic

or whatever other topic precedes this topic in the depth-first sequential ordering.

11th - (F11) **HELP *NEXT**
The current position in the Built-in Help database is move forwards along the depth-first sequential pathway. In other words the current topic's next topic is displayed. This may be a sister topic or an aunt topic (a sister topic to the mother topic) or a daughter topic.

12th - (F12) **RETRIEVE**
The first time that this PF key is pressed, it retrieves to the command line the name of the Built-in Help topic currently being displayed. If pressed a second time, the name of the next previously displayed HELP topic is displayed.

This PF key can be pressed repeatedly until the name of the desired topic is displayed. Then press ENTER to actually display that topic.

If you have taken a digression by following a crosslink from a prior topic, then using this **RETRIEVE** PF key is a good way to return to that prior topic.

Help FULlscreen Retrieve

As part of its fullscreen support, Z/XDC provides a command retrieval service. When you type a command, Z/XDC saves a copy of the command in a retrieval stack. Later, you can have any recently typed command retrieved from the stack and displayed again on the command line. You can then correct or otherwise adjust the command and then have it re-executed by pressing the ENTER key.

Z/XDC's command retrieval support has the following features:

- Each window on a display screen has its own command retrieval stack.

- A command retrieval stack can hold up to 255 previously issued command strings.

- Nearly all command strings issued for a window are saved in that window's command retrieval stack. The exceptions are:

  - Command strings that are generated as a result of a shortcut. (See HELP SHORTCUTCMDS for more information.)

  - Command strings that are generated from PF keys that are not modified by command line content. (See HELP FULLSCREEN PFKEYS HYPHEN for more information.)

- Duplicate command strings are **NOT** saved in the command retrieval stacks. Instead, when a duplicate is found of a command string that otherwise would be added, the already saved instance is moved to the top of the stack such that the next **RETRIEVE** command will retrieve it first.

- A **RETRIEVE** command can be used to retrieve saved command strings. The command can be used to move either backwards **or forwards** through the retrieval stack.

- The **RETRIEVE** command can be issued either from a PF key or from a command line as the last command in a usually complex string of commands. (See below for an example.)

- A **RETRIEVE LIST** command exists that produces a fullscreen display of the command retrieval stack. You can then do the following:
  - You can directly select the command string to be reissued.
  - You can edit a saved string to "fix" it within the commands retrieval stack.
  - You can delete a command string from the stack.

- When displaying Built-in Help topics, the **RETRIEVE** command takes on a special meaning: Instead of retrieving a previously typed command, it retrieves the name of a previously displayed HELP topic. Thus, **RETRIEVE** provides a means of retracing your steps through the Built-in Help database. For more information, see HELP ABOUTHELP.

As noted above, Z/XDC permits the **RETRIEVE** command to be issued from the command line. This can be particularly useful when it is appended to the end of a complex line of commands that you will want to reissue repeatedly but you don't want to go to the trouble of assigning to a PF key. Consider the following example:

- Suppose that you want to map the first 10 entries of a queue of control blocks. To do this you will have to load a dsect map of a queue entry, make 9 clones of the map, then issue 10 USING commands to assign base addresses of each of the maps. This example shows an easier way to do this.

  Suppose that the queue entries are control blocks named "SCB"s. Suppose further that the first SCB is queued from a field of the TCB named "TCBSTAB". Suppose also that the SCB's chain field is named "SCBCHAIN" and is the first field in the SCB. Then the following commands will load, clone, and base dsect maps as described above.

  DMAP XDCMAPS.TCBFIX
  This reads a TCB map from disk.

  USING .TCBRBP 21C%
  This assigns the TCB map to represent the current task's TCB.

  DM .SCB
  This reads an SCB map from disk.

  DM SCB1 SCB;U .SCBCHAIN .TCBSTAB% F;F .SCBCHAIN
  This (a) makes a clone of the SCB map named "SCB1", (b) assigns SCB1 to map the location pointed to by TCBSTAB, and (c) formats the storage now mapped by SCB1.

  DM SCB2 SCB;U .SCBCHAIN .SCBCHAIN% F;F .SCBCHAIN;RETRIEVE
  This (a) makes a 2nd clone of the SCB map named "SCB2", (b) assigns SCB2 to map the location pointed to by SCB1.SCBCHAIN, (c) formats the storage now mapped by SCB2, and (d) retrieves the entire command string to the command line so that it can be modified and issued again.

  Now that the previous command has been executed and then retrieved to the command line, in order to clone, base, and display a third SCB, all you need to do is change "SCB2" to "SCB3" and press ENTER. You can then merely repeat this process for any number of additional queue entries as you care to map.

For more information about how Z/XDC resolves address expressions that start with leading dots (.), see HELP ADDRESSING RESOLUTION.

Help FULlscreen SCrolling

Z/XDC provides for scrolling through two distinct objects - through a window's Scrollable Messages, and through storage.

For information about scrolling through **storage,** see **HELP FULLSCREEN SCROLLING STORAGESCROLLING.** For information about a window's **Scroll Area,** just read on.

Window Types

Z/XDC's terminal display area can be dynamically subdivided into one or more **windows** by the simple press of a PF key. (See PF key shift-F1 in **HELP FULLSCREEN PFKEYS DEFAULTKEYS SET-B.)** Each window:

- Is headed by a **Command Line.**
- Contains a **Display Area.**
- Maintains a **Scroll Area** of messages that can be brought into display via various scrolling commands, such as:
  - **UP**
  - **DOWN**
  - **LOCATE**
  - **SCANLOG**

For more information about **windows,** see **HELP FULLSCREEN WINDOWS.**

When the display is subdivided into multiple windows, the windows fall into two categories:

- The top window is the **Primary Window.**
- All other windows are **Watch Windows.**

The scroll areas for the two window types are significantly different from each other.

The Scroll Area, in General

Whenever any command is issued (including storage display commands), all of the messages generated by that command are appended to the bottom of a Scroll Area, and then the display window is positioned to show at least the starting portion of those messages.

If more messages are generated than can be displayed, the overflow is indicated by the appearance of two asterisks **(**)** at the bottom left-hand corner of the window, and you can use the **DOWN** scrolling command to get to them.

For Watch Windows, older displays are discarded, but for the **Primary Window,** older displays are pushed up the scrolling area and can be reached by **UP** and other scrolling commands.

The beauty of the **Primary Window's Scroll Area** is that it makes your debugging session's history instantly accessible! You simply use standard commands to navigate upwards, downwards or directly to any point you like. the navigation can be:

- By display lines (UP/DOWN HALF/FULL/DATA, UP/DOWN to cursor position, UP/DOWN a distance, etc.),
- By commands issued (UP C, UP/DOWN/LOCATE #cmdseqnumber),
- Or by entry from your program into Z/XDC (UP/DOWN TRACE).

The Scroll Area for the Primary Window

For the **Primary Window,** when a command string is executed...

- The command line is erased (making it ready to accept a new command string).
- All messages generated by the string of commands are appended to the bottom of the window's Scroll Area.
- The display is positioned to the first of those new messages.

- If more messages were generated than can be shown in the display area, two asterisks **(**)** are placed at the bottom left of the last displayed message. Excess messages can be brought into view via any scrolling command **(UP, DOWN, LOCATE** and **SCANLOG).**

- Prior messages are preserved in the scroll area, and they can be reached via **UP** and **LOCATE** commands.

- All generated messages are written into a **Session Log** which usually is located on JESn Spool.

- The Scrollable Area (by default) can hold approximately **10,000** messages, but this is user settable. (See **HELP COMMANDS SET LOG MANAGEMENT.)**
- Excess messages are purged from the Scroll Area (but remain in the spooled Session Log, of course).

The Scroll Area for Watch Windows

For **Watch Windows,** when its command string is executed...

- The command line is **not** erased. Instead, the command string is edited (to remove action commands) and the retained displaying-type commands are left to be re-executed automatically upon every press of the ENTER key or of any PF key.
- All previously generated messages are discarded. They cannot be scrolled to.
- A new scroll area is built for all of the messages generated by the just executed string of commands.
- The display is positioned to the first of those new messages.

- If more messages were generated that can be shown in the display area, two asterisks **(**)** are shown at the bottom left of the last displayed message. These **Excess Messages** can be reached via **DOWN** and **SCANLOG** commands.
- The scrolling commands (UP, DOWN, LOCATE and SCANLOG) are **ineffective unless Excess Messages are present,** in which case the scrolling commands can be used to bring them into view.
- Scrolling commands **cannot** be used to bring prior displays into view (because those have been discarded).
- The generated messages are **not** written to the Session Log.

Latent Command Displays

Normally, the displays produced by **Latent Commands** do not appear in the Scroll Area. (They only appear in the Session Log.) But if you use the **SET LOG LATENT=SHOW** command, they will appear in the Primary Window's Scroll Area. For more information about Latent Commands, see **HELP FULLSCREEN LOGGING LATENTCOMMANDS.**

Watch Window Display Stability

If the previous set of displayed messages had been positioned downward via scrolling commands, then that downward positioning is retained. Thus, display positioning is stabilized. This is particularly useful when the number of generated messages far exceeds the size of a window's display area.

Downward positioning can be expressed relative either to the top or bottom of the generated messages:

- **DOWN nnn** sets the scroll position relative to the top of the generated messages.
- **DOWN MAX-nnn** sets the scroll position relative to the bottom of the generated messages.

Understanding what **DOWN MAX-nnn** does can be a bit tricky. This is because **DOWN MAX** does not cause the last line of the scrollable messages to be displayed at the top of the window's display area. Instead, it causes that line to be displayed at the bottom of the area. This means that issuing something like DOWN **MAX+6** (yes, **plus** 6) actually makes sense. Try it. You'll see what I mean.

This Display Stability facility does not apply to the Primary Window.

Primary Window - Navigating the Scroll Area

As noted above, the **UP, DOWN, LOCATE** and **SCANLOG** commands can be use to navigate the Primary Window's in various expected ways. There are also a couple of **unexpected** ways worth learning about, so keep reading.

The UP and DOWN command accept of course such typical operands as **MAX CURSOR PAGE FULL HALF DATA** and **nnn** so I won't bore you with those details here. They are quite adequately described at **HELP COMMANDS UP.**

But...

UP|DOWN|LOCATE #nnn
All command strings that you have issued on the top command line appear numbered in the Scroll Area. So you can that number with any of these commands to jump directly to any specific command string.

In the case of this **#nnn** operand, direction does not matter. Any of these commands can be used regardless of in which direction the Area has to be scrolled to reach command number #nnn.

UP|DOWN C|CMD

These commands will scroll the Scroll Area up or down to the next prior or next following command string.

Notice that in Z/XDC, **C** means **CMD** (not CURSOR).

UP|DOWN TRACE;RETRIEVE

T [in combination with F7 or F8]

After you've used the **TRACE** command to step execution through your program for awhile, you can then use these commands to "play a movie" of your program's execution so far.

The **UP TRACE** command (and DOWN TRACE as well) accomplish this by scrolling the Scroll Area up or down to the next prior or next following display that was the first display produced upon Z/XDC receiving control back from the user program. Usually, this display will have been generated by an automatically issued **WHERE** command showing the then current execution location.

If you have the default PF keys in effect, then you can run your movie just by repeatedly typing **T** on the command line and pressing **F7** or **F8.** (Or you can press the **Page Up** or **Page Down** keys, depending upon your keyboard setup).

Using Signed Offsets (+nnn and -nnn)

Regardless of which operand you use on the **UP** and **DOWN** commands, you can append one or more **signed offsets** to adjust the resulting positioning. So for example, **DOWN MAX+6-2** will position the Scroll Area so as to display four lines of whitespace following the last message stored in the Scroll Area.

When a scrolling operand results in a negative scrolling distance, then the UP or DOWN command processes as if it were the opposite command.

Notice, while DOWN MAX+nnn can make sense, the same is not true of UP MAX+nnn. While there is something south of DOWN MAX, there is nothing north of UP MAX.

When a scrolling operand results in a location that is beyond the range of the Scrolling Area, an **UP|DOWN MAX** command is silently used in its place.

UP|DOWN Commands and PF keys

The Factory Default PF keys contain several UP and DOWN commands:

F7: **UP -** - Scrolls upwards either according to the rules of **UP CURSOR** or by an amount given on the command line.

F8: **DOWN -** - Scrolls downwards either according to the rules of **DOWN CURSOR** or by an amount given on the command line.

shift-F7: **UP M** - Scrolls upwards to the top of the Scroll Area.

shift-F8: **DOWN M** - Scrolls downwards to the bottom of the Scroll Area.

Dual-Use DOWN Command

Normally, the DOWN command is used to scroll through the Historical Scroll Area. But sometimes, depending up context, the DOWN command can be used to scroll through your program's actual storage. In order for this to occur, **all** of the following must be true:

1st - The **DOWN** command must be issued from the **Primary Window.** (This doesn't work for Watch Windows.)

2nd - The **last** display recorded in the Scroll Area (i.e. the most recently issued command that generated messages) must have been an **[E]WHERE, FORMAT** or **DISPLAY** command.

3rd - The **DOWN** command must **not** have operands.
(Or you can just press **F8.)**

When all of the above are true, then under the covers, the DOWN command issues a **FORMAT** command (or DISPLAY command, if appropriate) **without** operands.

When no operands are given, these commands generate a display of the storage that immediately follows the last prior display. This gives the appearance of scrolling down through storage.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

STORAGESCROLLING - Simulating scrolling through storage.

Help FULlscreen SCrolling Storagescrolling

Display Currentness

When you issue a **FORMAT** or **DISPLAY** [or etc.] command to display storage, no matter how Z/XDC implements the display, by the time you see it, it already is ancient history. So for example, if you attempt to display a highly active location (such as low code, aka the PSA) even in the time it takes for your mind to perceive what you are looking at, the values show in many fields will no longer be current. This is not to say that such displays are useless. This is just something you need to keep in mind, that's all.

Fortunately, although the above scenario is quite real, it also is rather extreme. In most real world scenarios with which you work, the code and data that you will be examining with Z/XDC will be quite stable.

The Display Construction Process

So when you issue a command to show storage, here is what happens:

- First, Z/XDC extracts into its own buffer a copy of the storage you've asked Z/XDC to display.
- Then Z/XDC formats the storage data into several text messages according the command that you have issued.
- Then the messages are appended at the bottom of the scrollable messages queued to the Window from which you issued the storage display command.
- Then the Window's scroll area is positioned such that the first of the generated messages will appear in the Window's first display line.
- Finally, the display is sent to your terminal for you to view.

So, what does all this mean? It means that when you see a storage display, you really are viewing the **bottom of a session log** showing the most recently generated messages from Z/XDC commands.

The consequence of this is that when you issue a scrolling command **(UP, DOWN** and friends), you are scrolling through your Session Log, not through storage. **But** please read on...

Scrolling DOWN Through Storage

So while strictly speaking, Z/XDC can scroll up and down only through the session log, if you want to scroll down through storage, Z/XDC has a pretty effective way to simulate it. Here's what happens.

When you issue a **DOWN** command:

- If you issue it from the **Primary Window,**
- And if the bottom messages in that Window's Scroll Area are from a previously issued storage displaying command **(FORMAT WHERE DISPLAY** etc.),

Then Z/XDC will issue a storage displaying command such that it will produce a display of the next chunk of storage formatted in the same was as the prior command.

Scrolling UP Through Storage

Unfortunately, Z/XDC does not have support for scrolling up through storage. (The **UP** command will only scroll up through the session log.)

If you wish to scroll up and down through storage at will, then one thing you can do is issue a **DISPLAY** or **FORMAT** command with either a **LINES=nnn** or **BYTES=hhh** operand to generate a large formatted display of storage into the Session Log. Then you can scroll up and down through that.

Help FULlscreen Windows

Z/XDC's fullscreen support permits you to subdivide your physical terminal display screen into **two or more** individual display areas. Each such area is referred to by Z/XDC as either a **Primary Window** or a **Watch Window:**

- Each window has its own **command line** where command strings can be typed.
- And each window has zero or more **display lines** wherein the responses to its commands are displayed.
- The number of display lines available to a window depends upon the **locations** of the command lines on your terminal's display screen.
- Each window has its own retrievable, 255 slot record of previously issued command strings. The **RETRIEVE** command can access this either forwards or backwards or as a selection list. For more information, see HELP FULLSCREEN RETRIEVE.

Window Types

When your terminal's screen is divided into two or more windows, the top window is always called the **Primary Window.** All additional windows, if any, are referred to as **Watch Windows.**

The **Primary Window** is where you will issue the bulk of your Z/XDC commands. For more information, see HELP FULLSCREEN WINDOWS PRIMARYWINDOW.

A **Watch Window** is a Window that you can create or destroy with simple presses of PF keys, and with which you can issue information and storage displaying commands of all sorts. They are intended for persistent displays of whatever you like. For more information, see HELP FULLSCREEN WINDOWS WATCHWINDOWS.

Example:

Suppose that you have divided your terminal screen into three windows. Then the **top window** is your Primary Window, while the **bottom two** are Watch Windows.

Suppose then that you issued a **LIST REGS** command from one of the Watch Windows and a **FORMAT .MSGBUFR** command from the other. (This assumes, of course, that **MSGBUFR** is a suitably defined label in one of your data areas.)

These two command strings will automatically be saved and reissued each time an entering key is pressed.

In particular, if you then use the **T** or **T BY** command (via the Primary Window) to trace a series of user program instructions, and if those instructions cause the registers and/or the **MSGBUFR** data area to change, then those changes are displayed as they occur! In fact, it's almost as if you are **watching a movie** of the work being done!

A Caution About the Current Display Pointer

The Primary Window and each Watch Window maintains its own **Current Display Pointer (CDP** - see HELP ADDRESSING IMPLICIT for more information).

This means that commands that reference or set the CDP issued from one window will not affect the CDP setting for another window. (That's good.)

On the other hand, **@CDP** is a built-in equate that labels the location that the CDP currently points to. (See HELP EQUATES BUILTIN OTHER for more information.) Well, since each window effectively has its own CDP, the location of the @CDP equate shown in one window will be unrelated to the CDP's location in a different window. If you allow yourself to become confused by this, hilarity will ensue.

Help FULlscreen Windows Primarywindow

The **Primary Window** is the primary window in which you will do the bulk of your debugging work and issue most of your Z/XDC commands. The Primary Window always starts with the display's first command line (which is always on the display's second row), and its display area proceeds down either to the next command line or the bottom of the display if there are no more command lines.

The Primary Window has the following characteristics that are different from Watch Windows.

- The Primary Window always exists. It cannot be removed.
- Any Z/XDC command can be issued from its command line.
- Upon pressing the ENTER key, whatever command string is present on the command line is processed, and the command line is erased in preparation to accept whatever new command string you care to type.
- All commands issued from a Primary Window and all display messages sent to a Primary Window are written to a permanent Session Log which usually is allocated on spool. This Session Log is very useful for post-mortem examination.
- Up to the most recent 10,000 (or so) messages sent to the Session Log are also saved in a Scroll Area that can be perused via Z/XDC's scrolling commands (UP DOWN LOCATE and SCANLOG).
- When the auguries are right, the **DOWN** command can be used to scroll through storage (not just through the Scroll Area). See **HELP FULLSCREEN SCROLLING STORAGESCROLLING** for details.
- When a fullscreen action (such as a Point-and-Shoot command) results in a display, that display is always shown in the Primary Window. This is true even when the action occurs within a Watch Window.

Help FULlscreen Windows WAtchwindows

Watch Windows are secondary windows that can be created and deleted by simple presses of PF keys. (See PF key shift-F1 in HELP FULLSCREEN PFKEYS DEFAULTKEYS SET-B.)

Watch Windows are intended for persistent displays of whatever you like. They have both limitations and advantages with respect to Primary Windows. The differences are these:

- Watch Windows can be created and deleted dynamically just by pressing a PF key. Assuming Factory Defaults, the PF keys to use are:
  - **shift-F1** creates Watch Windows. Just move the cursor into the display row at which you want a new command line created, and press PF key shift-F1. A new command line will pop into existence, and it will contain a copy of either:
    - The next higher Watch Window's command string,
    - Or the next lower Window's command string, if there is no higher Watch Window.
  - **F1** deletes Watch Windows. Just move the cursor into any row of the Watch Window you want to remove, and press PF key F1.

- When you type a string of commands onto a Watch Window's command line, an edited copy of that string is retained and re-executed every time either the ENTER key or any PF key is pressed. So Watch Windows are an excellent device for presenting persistent displays.

- Any Z/XDC command can be issued from a Watch Window's command line. ***BUT*** after the command string has been executed for the first time, all action commands are removed from the string and only displaying type commands are retained for re-execution. Retainable commands include:
- * [an asterisk]
- DISPLAY
- EWHERE
- FORMAT
- LIST
- SHOW
- WHERE

The messages displayed in a Watch Window are **not saved** in a session log. They are simply discarded.

The messages also are not saved in a scroll area, so they cannot be reached by UP and LOCATE commands. UP, DOWN, LOCATE and SCANLOG will only work within the messages generated for the current execution of the Window's command string.

The display screen's current layout, i.e the locations and contents of all Watch Window command lines, can be saved in a named Session Profile via the **PROFILE SAVE name** command.
- Because the profiles are named, it is quite reasonable to save multiple profiles each having its own display layout.
- Any named profile can be easily reloaded with a simple **PROFILE READ name** command.

Help MACros

Included with the Z/XDC product distribution is a macros library containing a rather large number of Assembler macros that customers may find useful in their own programming. They can be found in the **XDCMACS** library. Its factory default name is hlq.XDCV31.XDCMACS, but please check with your SysProg for the library's actual name on your System.

The XDCMACS library contains over 300 macros! But most of them are provided only in support of the various sample Assembler programs that we ship in the **XDCASM** library. They will not be further discussed here. You are welcome, of course, to examine the source of any macro, as the urge occurs. You are also welcome to contact us with any questions you might have.

On the other hand, several of the XDCMACS macros were written specifically to aid in the use of Z/XDC. Those are summarized in subtopics.

All of these more useful macros contain comprehensive commentary documenting both the purpose of the macro and its detailed usage syntax, so there is no need to go into that kind of detail here.

Subtopics Index

Please review the following subtopics for more information on selected macros. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

#cblocks - This is a series of a couple of hundred **envelope** macros that provide a consistent way to invoke IBM mapping macros for z/OS control blocks.

#DBCPARM - This macro maps a parameter block (DBCPARM) used in Z/XDC's API for several user exits.

#DBCR0IN - This macro maps the flags to set into R0 when a user written recovery routine makes a direct call to Z/XDC.

#DBCR0UT - This macro maps the flags that Z/XDC passes back in R0 when it returns to its direct caller (a user written recovery routine).

#DBCVRSN - This is a stage setting macro for Z/XDC related assemblies.

#DIE - This macro traps illogical conditions and passes control to Z/XDC (if present).

@SYMOFF/ON - Use these macros to remove unwanted ranges of SYM data and ADATA from the output files created by the Assembler.

#TEST - This macro performs a host of assembly time services, including the flagging interdependency violations.

#XDCHOOK - This macro inserts a static hook into your code for creating a debugging session on the fly.

#XDCLOC8 - This macro finds Z/XDC's primary load module (named **XDC)** when it resides in common storage (PLPA or DLPA). It does this without using z/OS's LOAD service.

Help MACros #Cblocks

The **#cblock** macros are a couple of hundred mapping macros for IBM control blocks. They are envelope macros for invoking native IBM mapping macros in a consistent way.

I wrote these **#cblock** macros (a) to provide a consistent naming convention (#DCB, for example), and (b) to hide the inconsistent variety of operands that are required when invoking the IBM macros natively.

These macros are included mainly to support the 26 **SRCMAPSx** assemblies found in the **XDCASM** library. See HELP MAPS XDCMAPS for more information about those assemblies.

Help MACros #DBCParm

What is (and Where is) the #DBCPARM Macro?

The **#DBCPARM** macro generates a parameter block named **DBCPARM.** It is optional, but if you wish to provide exit routines (or other customizations) for use by Z/XDC, you will need to create a **DBCPARM** to do so.

The **#DBCPARM** macro can be used to create either a map of **DBCPARM** or a data area into which a **DBCPARM** can be built.

The **#DBCPARM** macro is available in the **hlq.XDCV31.XDCMACS** Product Library. You will have to ask your Systems Programmer for the library's actual name at your Data Center.

Passing a DBCPARM cblock to Z/XDC

When Z/XDC receives control as a recovery routine (ESTAE[X], ARR, FRR, etc.), **GR2** either will point to DBCPARM or will equal zero.

If Z/XDC receives control in **AR mode,** AR2 must contain the ALET needed to access DBCPARM.

When Z/XDC is called by z/OS's RTM, GR2/AR2 will be as just described.

When you need RTM to pass a DBCPARM block to Z/XDC, you must create the block yourself and make it known to RTM by one or another of the following mechanisms.

When Z/XDC is directly called by a user written recovery routine, it must set up GR2/AR2 as just described. (See **HELP DEBUGGING ESTAES MODIFYING** for further details.)

Setting Up DBCPARM When Z/XDC is Running as an ESTAE or ESTAEX

Use the **PARAM=** operand of the **ESTAE** or **ESTAEX** macro that sets up Z/XDC. Example:

```
        LOAD   EPLOC==CL8'XDC',
               ERRET=NOXDC
        ESTAEX (R0),PARAM=DBCPARM
```

Setting Up DBCPARM When Z/XDC is Running as an ESTAI

Use the second subfield of the **ESTAI=** operand of the **ATTACH** macro that sets up Z/XDC as an **ESTAI.** Example:

```
        LOAD  EPLOC==CL8'XDC',
              ERRET=NOXDC
        LR    R2,R0
        ATTACH EP='MYSUBPGM',
              ESTAE=((R2),DBCPARM),...
```

Setting Up DBCPARM When Z/XDC is Running as an FRR

Use the first word of **FRRSPARM** when Z/XDC has been set up as an **FRR.** Example:

```
        LOAD  EPLOC==CL8'XDC',
              ERRET=NOXDC
        SETFRR A,               Create FRR
              FRRAD=(R0),       @'XDC
              PARMAD=(R3),      Receives @'6-word parm area
              WRKREGS=(R14,R15),
              MODE=FULLXM       Whatever
              EUT=YES           Whatever
        LA    R0,DBCPARM
        ST    R0,0(R3)
```

Setting Up DBCPARM When Z/XDC is Running as an ARR

If you have setup Z/XDC to run as an **ARR,** then you need to use an **MSTA** instruction to store **DBCPARM's** 31-bit address and 32-bit ALET into the 8-byte **modifiable area** for the PC routine for which the ARR routine is established.

Issue the MSTA first thing, near the start of your PC routine.

Building a DBCPARM Cblock

The **DBCPARM** control block is entirely optional. If one is provided, then it must be doubleword aligned in storage.

If one is **not** provided, then Z/XDC simply proceeds without its services.

To find out whether or not a **DBCPARM** block has been provided, Z/XDC proceeds as

follows:

- Z/XDC starts by checking the **GR2/AR2** registers that it received from its caller (RTM or a user written recovery routine).
- If **GR2** is **nonzero,** Z/XDC then checks for certain "eye-catcher" fields. If they are present and accessible, then Z/XDC proceeds to use the DBCPARM.
- Otherwise, Z/XDC concludes that GR2/AR2 does **not** point to a DBCPARM, so it proceeds without its services.

Z/XDC's support for **DBCPARM** is completely **downward compatible.** Z/XDC recognizes all older versions of **DBCPARM** going back several decades.

Z/XDC Requirements for Accessing a DBCPARM Block

IMPORTANT! Z/XDC always executes in **Primary** Address Space Control Mode (ASC-MODE) in the **Home** Address Space with the Primary Address Space set equal to the Home Address Space. This means that both the **DBCPARM** control block and anything that it points to must reside in a space that is accessible to Z/XDC when running in this mode.

If the DBCPARM block's location is **ALET qualified,** then Z/XDC will switch into AR ASC-MODE when accessing it. So the ALET must be valid for use in the **home** address space. This means:

- For space switched environments (such as space switching PC routines), **DBCPARM** and its dependents will most likely have to reside in shared storage (CSA, PLPA, SQA, etc.), so **DBCPARM's ALET** should probably =0.
- For nonspace-switched environments, **DBCPARM** may be either in the home space or in any other address space or dataspace that can be reached via ALET from the home space.

The Major Services Provided by #DBCPARM

The **#DBCPARM** macro and the **DBCPARM** control block provide several major services:

- The **DBCPARM** control block is required for communicating to Z/XDC the specific **user exits** and other customizations that you wish to provide.
- The **#DBCPARM** macro provides extensive **commentary** describing in detail how to code for several of Z/XDC's user exit interfaces.
- The **DBCPARM** control block provides several **communication fields** needed for various user exit routines.
- The **#DBCPARM** macro contains several Assembler Language **equate symbols** that identify interface constraints, characteristics, and capabilities.

Z/XDC Exits Supported by the DBCPARM Control Block

The **DBCPARM** control block allows you to provide the following exits and other information to Z/XDC:

Clone Name

If Z/XDC has been renamed from **XDC** to some other 3-character clone name, then the **DBCPARM** control block provides a way for Z/XDC to know what its name is. The clone name has to be set into the control block's **DBCPBID** field.

Note, when debugging space switching PC routines, this **DBCPBID** field is the **only** way for Z/XDC to know what its name is. For more information, see **HELP DEBUGGING PCROUTINES.**

In other debugging scenarios, Z/XDC can discover the name of the load module it is running as, and take its clone name from that.

UCI Exit

You can provide a **User Communications Intercept** exit routine to take over all communications between Z/XDC and the display terminal. When a **UCI** exit is present, Z/XDC uses it, instead of a terminal, as its user communications "device". This routine can be used to send commands to Z/XDC and to receive messages and displays back from Z/XDC. See **HELP EXITS UCI** for more information.

- Use the **DBCUCIAD** field to provide the address of the exit.
- Use the **DBCUCIUD** field to maintain for your UCI the address of an exit Provided Data Area.

When Z/XDC is used to debug a space switching PC routine, both the **DBCUCIUD** exit routine and its data area (if any) must be located in shared storage (CSA, PLPA, SQA, etc.). See **HELP DEBUGGING PCROUTINES** for more information about debugging PC routines.

FRREND Exit

You can provide an exit that is called only when:

- Z/XDC is running as an **FRR** routine.
- **And** you issue an **END** command to end the debugging of the current SRB mode routine or FRR protected task mode routine.

This exit can be used:

a) To control the manner in which the current routine is ended, and
b) To notifying programming that the SRB (or FRR-protected RB routine) is ending.

The **DBCPARM** control block provides both:

- The address of the exit (in the **DPFRREND** field)
- And a 4-byte value to pass to the exit (in the **DPFRREUD** field).

See **HELP EXITS FRREND** for more information.

Miscellaneous Exits

Many (but not all) of Z/XDC's exits share a common router. When Z/XDC calls one of these exits, it calls the router and provides an exit type code so that the router knows which exit is being called.

A sample router, named **XDCXITS** is provided in Z/XDC's **XDCLINK** load library (default name is **hlq.XDCV31.XDCLINK).** The source for **XDCXITS** is provided in the **hlq.XDCV31.XDCASM** library. For more information, see **HELP EXITS MISCXITS.**

The Miscellaneous Exits Router supports the following exits:

- Initialization Exit
- Resumption Exit
- Termination Exit
- Trace User SVC Exit
- User Commands Exit
- User Register Stack Exit

The **DBCPARM** control block provides the addresses both
- Of the **XDCXITS** exit router (in the **DBCXITSX** field)
- And of numerous data communications fields needed by the various exits supported by the router.

For detailed information, see **HELP EXITS exitname.** Also, read the extensive commentary located in the **#DBCPARM** macro located in the **hlq.XDCV31.XDCMACS** library.

When Z/XDC is used to debug a space switching PC routine, both the Miscellaneous Exits Router (if any) and all required data areas must be located in shared storage (CSA, PLPA, SQA, etc.). See **HELP DEBUGGING PCROUTINES** for more information.

An Initial Commands String

You may use the **DBCPARM** control block to provide the address of a string of Z/XDC commands to be executed when Z/XDC is first called at the start of a debugging session. Just set **DBCCMDAD** to the address of the string.

The string must be null-delimited (i.e. ended by an **X'00'** value), and it may be up to 16 megabytes long. See commentary in the **#DBCPARM** macro for more information.

When Z/XDC is used to debug a space switching PC routine, the command string must be located in shared storage (CSA, PLPA, SQA, etc.). See **HELP DEBUGGING PCROUTINES** for more information.

A Reoccurring Commands String

You may use the **DBCPARM** control block to provide the address of a string of Z/XDC commands to be executed **every time** that Z/XDC is called during a debugging session. Just set **DBCCMDA2** to the address of the string.

The string must be null-delimited (i.e. ended by an **X'00'** value), and it may be up to 16 megabytes long. You may change the command string at any time according to your needs. Again, see commentary in the **#DBCPARM** macro for more information.

When Z/XDC is used to debug a space switching PC routine, the command string must be located in shared storage (CSA, PLPA, SQA, etc.). See **HELP DEBUGGING PCROUTINES** for more information.

Debugging Session Scope

You can set a flag **(DPBFMYOB)** that controls whether the scope of the debugging session is address space wide or only task-wide.

When the scope is only task-wide, then all breakpoints, maps, equates etc. created by a copy of Z/XDC running under one task will be completely unknown to another copy running under another task.

See commentary in the **#DBCPARM** macro for more information.

XDCCALL[A] Uses DBCPARM - Here's What it Does With it

If you start your debugging session using the **XDCCALL** utility (or any of its aliases), then XDCCALL automatically builds a **DBCPARM** block and passes it to Z/XDC proper. For more information about XDCCALL, see **HELP XDCCALL.**

XDCCALL (and friends) builds a DBCPARM to implement the following services:

- **Initial Commands Script:** XDCCALL allows you to provide an Initial Commands Script by allocating it with a ddname of **XDCCMDS.** If **XDCCALL** finds a **//XDCCMDS** allocation, it then determines the DSNAME of the dataset that is pointed to by that allocation. Then it builds a **READ dsname** command string and stores a pointer to that string into the **DBCCMDAD** field of the **DBCPARM** control block. This causes the script to be read and processed at the very start of the debugging session.

- **User Communications Intercept Exit:** XDCCALL supports providing a **UCI** exit. If you create a load module named **XDCUCI** and place that module into any library that is a part of your current task's load module search order (//TASKLIB, //ISPLLIB, //STEPLIB, //JOBLIB, the PLPA or a System linklist library), then **XDCCALL** will load that module into storage and place its address into the **DBCUCIAD** field of the **DBCPARM** control block. For more information, see **HELP EXITS UCI.**

[You can use the **LIST SEARCHORDER** command to show the load module search order for any task running in any address space (security permitting) in the system.]

- **Miscellaneous Exits Interface Router:** XDCCALL supports providing a Miscellaneous Exits Router. If you create a load module named **XDCXITS** and place that module into any library that's in the current task's load module search order, then **XDCCALL** will load that module into storage and place its address into the **DBCXITSX** field of the **DBCPARM** control block. For more information, see **HELP EXITS MISCXITS.**

  Note, a sample **XDCXITS** module is provided in the **hlq.XDCV31.XDCLINK** load library. (Ask your Systems Programmer for the library's actual name at your Data Center.)

  The source code for this sample can be found in the **hlq.XDCV31.XDCASM** library.

  This sample implements a user command named **LIST TIOT.**

For more information about providing exits to Z/XDC, see **HELP EXITS.**

Help MACros #DBCR0In

What is (and Where is) the #DBCR0IN Macro?

The **#DBCR0IN** macro may been needed if you are writing your own recovery routine that will (or might) BASR to Z/XDC as a subroutine. You will need **#DBCR0IN** if you intended to make **informational** or **conditional** calls to Z/XDC. It maps the settings you will need to put into R0 when you make such calls.

See **HELP DEBUGGING ESTAES MODIFYING** for why and how you would integrated Z/XDC into your own recovery routines and what the three call types are.

#DBCR0IN generates a map of the control flags you may have to set into **R0** before BASR'ing to Z/XDC.

The **#DBCR0IN** macro is available in the **hlq.XDCV31.XDCMACS** Product Library. You will have to ask your Systems Programmer for the library's actual name at your Data Center.

More Information

For more information, see:

- **HELP MACROS #DBCR0UT**
  Describes the **#DBCR0UT** macro, what it is and where to find it.

- **HELP DEBUGGING ESTAES MODIFYING**
  Explains why you might want to put a Z/XDC interface into your recovery routine and how to do it.

- **HELP DEBUGGING ESTAES MODIFYING INFORMATIONALCALLS**

Details the outputs returned when Z/XDC makes its return to your recovery routine.

Also, read commentary within the **#DBCR0IN** and **#DBCR0UT** macros, themselves. The information provided there is comprehensive!

Help MACros #DBCR0Ut

What is (and Where is) the #DBCR0UT Macro?

The **#DBCR0UT** macro may been needed if you are writing your own recovery routine that will (or might) BASR to Z/XDC as a subroutine. You will need **#DBCR0UT** if you intended to make **informational** or **conditional** calls to Z/XDC because it maps the flags that Z/XDC sets into **R0** before returning from such calls.

Note, the same flags are set into **R0** when Z/XDC is returning from a debugging session that is being ended by the **END** command.

See **HELP DEBUGGING ESTAES MODIFYING** for why and how you would integrated Z/XDC into your own recovery routines and what the three call types are.

#DBCR0UT generates a map of the control flags you will receive in **R0** when Z/XDC returns to your recovery routine.

The **#DBCR0UT** macro is available in the **hlq.XDCV31.XDCMACS** Product Library. You will have to ask your Systems Programmer for the library's actual name at your Data Center.

More Information

For more information, see:

- **HELP MACROS #DBCR0IN**
  Describes the **#DBCR0IN** macro, what it is and where to find it.

- **HELP DEBUGGING ESTAES MODIFYING**
  Explains why you might want to put a Z/XDC interface into your recovery routine and how to do it.

- **HELP DEBUGGING ESTAES MODIFYING INFORMATIONALCALLS**
  Details the outputs returned when Z/XDC makes its return to your recovery routine.

Also, read commentary within the **#DBCR0IN** and **#DBCR0UT** macros, themselves. The information provided there is comprehensive!

Help MACros #DBCVrsn

#DBCVRSN is an inner macro required by many other Z/XDC macros. It's a stage setting macro for Z/XDC related assemblies. Among other things, #DBCVRSN sets (and documents) a number of assembly time **GBLx** symbols and **EQU** equate names. If you're curious, you're welcome to review its internal commentary.

In addition, the macro has a commentary section that lists every person who has made a contribution (some large, some small) to the code.

For more information about some of the contributors, see **HELP SUPPORT CREDITS.**

Help MACros #DIe

The **#DIE** macro generates what we call a **dead trap.** This can be used at assembly time to place an intentional trap (breakpoint) to Z/XDC. The trap can convey either a message to be displayed or a Z/XDC commands string to be processed. For more information, see:

- HELP BREAKPOINTS DEADTRAPS
- HELP BREAKPOINTS DEADTRAPS #DIE

#DIE can be used to vcheck dependencies and detect illogical conditions at **run time.** See **#TEST** for detecting dependency violations at assembly time.

The #DIE macro source contains comprehensive usage and syntax commentary that you need to review before using this macro.

Help MACros @symoff/on

The @SYMOFF and @SYMON Macros

These macros emit the controls necessary to purge from Assembler output files both SYM data and ADATA that is inconvenient and/or excessive. They accomplish this by emitting Assembler statements that trigger removal and resumption actions by post-assembly filtering programs, **xxxSYMED** and **xxxADATA.** Briefly, the macros do the following:

- **@SYMOFF** macro calls emit **SYMDEL DSECT** statements.
- **@SYMON** macro calls emit **SYMUNDEL DSECT** statements.

Both macros then issue **&SYSLOC LOCTR** statements to immediately restore whatever the prior xSECT was. This allows the macros to be used both **nondisruptively** and **repeatedly** pretty much anywhere it makes sense to.

(One place where it does **not** make sense to would be at the start of your program, prior to all xSECT statements.)

The xxxSYMED and xxxADATA Filtering Programs

(Note, **xxx** refers to Z/XDC's current **clone** name. Usually, that will be **XDC,** but it could be any legal 3-character name. See **HELP XDCCLONES** for more information.)

The xxxADATA and xxxSYMED programs are independent utilities that can be added to an assembly proc to filter unwanted records from ADATA and SYM data outputs (respectively) produced by the Assembler:

- **xxxADATA** will remove unwanted ADATA either from GOFF object files or from the ADATA side files that the Assembler writes to //SYSADATA. For details, see **HELP MAPS ADATA EXCESSADATA.**

- **xxxSYMED** will remove unwanted SYM data either from classic object files or from GOFF object files. For details, see **HELP MAPS SYM EXCESSSYMDATA.**

If you set up your Assembler to emit **both** SYM data and ADATA, then you can add both of the filtering utilities to your assembly proc to process both the SYM data and ADATA outputs.

The SYMDEL, SYMUNDEL and SYMNODEL Triggers

@SYMOFF and **@SYMON** emit the **SYMDEL** and **SYMUNDEL** filtering triggers, respectively. No macro has been written to emit the SYMNODEL trigger. If desired, SYMNODELs have to be added to your code manually.

- SYMDEL and SYMUNDEL work as **nesting pairs.** When a filtering program encounters one or more SYMDELs, all subsequent SYM/ADATA records are scrubbed from their output files until an **equal number** of SYMUNDELs is encountered.

- SYMNODELs terminate filtering no matter how many unmatched SYMDELs remain. It also resets the nesting counter to zero.

Using the @SYMOFF and @SYMON Macros

These macros exist purely both for convenience and for reducing clutter in assembly listings:

- Without these macros, you would have to insert two statements at every trigger/untrigger point:
  - SYMDEL DSECT ,
  - whatever LOCTR ,

- The @SYMOFF/ON macros will issue those statements for you, and they will do so **silently** thereby reducing listing clutter to a minimum. (If you really want to see what gets emitted, throw in a PRINT OPSYN ANOP statement just ahead of your macro call.)

With the reduced clutter, it is quite easy to use these macros as frequently as you like throughout your code.

The macros do not need either name fields or operands (mostly). So if you try to provide such, they will be ignored (mostly). There is an exception:

- If you try to use these macros in nonsensical places, they will throw a **level-1** MNOTE and then abort. This will cause the MNOTEs to show up in **//SYSTERM.** It will also set the Assembler's minimum completion code to **1.**

- If you wish to avoid both of these consequences, you can use **QUIET** as an operand. This will cause the MNOTE to be changed from a level-1 MNOTE to an **unleveled** MNOTE.

Examples: Suppose you placed your first @SYMOFF macro at the start of your source prior to all CSECT, DSECT and RSECT statements. Well, that's a no-no, so the macro will issue an error message and then abort. These examples show how the **QUIET** operand affects the emitted error message.

@SYMOFF ,
@SYMOFF WHATEVER JUNK YOU LIKE
For both of these examples, an error message would be issued as a **level-1** MNOTE. (Note that unrecognized junk operands are ignored.)

@SYMOFF QUIET
In this example, the error message would still be issued, but this time as an **unleveled** MNOTE.

Example: Uncluttering reORG'd Data in a Translate Table

One technique that Assembler programmers sometimes use is to define a background data area or machine instruction and then use **ORG** statements to overlay the background with custom operands (for machine instructions) or altered data (for a translate table). For Instance...

```
       DS  0D               | [alignment]
UPCASE DC  256AL1(*-UPCASE) V [background]
       @SYMOFF ,               [suppress map data]

       ORG UPCASE+C'a'         [locase a-i]
       DC  C'ABCDEFGHI'        [upcase A-I]

       ORG UPCASE+C'j'         [locase j-r]
       DC  C'JKLMNOPQR'        [upcase J-R]

       ORG UPCASE+C's'         [locase s-z]
       DC  C'STUVWXYZ'         [upcase S-Z]

       ORG ,                   [resume high]
       @SYMON ,                [resume map data]
```

When you issue a **FORMAT .UPCASE** command, the UPCASE translate table will display fairly nicely in 8 rows as a single, cohesive field.

On the other hand, if the @SYMOFF and @SYMON macros were removed, then the formatted display would be cluttered and broken up by the reORG'd subfields. All in all, it would be quite a bit harder to apprehend.

Example: Uncluttering reORG-Constructed Machine Instructions

Here is an example of a macro generated machine instruction (OIHF) whose operands are manually built from several macro-time computed values:

- First, a template instruction is emitted.
- Then an ORG *-4 is emitted so that the immediate operand can be rebuilt.
- Then several DCs are emitted to manually rebuild the immediate operand per variable, custom needs.

```
         OIHF RH0,*-*           template
OIHF     @SYMOFF ,              Suppress mapping data
         ORG  *-4               overlay immediate field
         DC   AL1(&NFVL)        value for ix n/f
         DC   AL1(&NCVL)        value for ix n/c
         DC   AL1(0)            index (already set)
         DC    AL1(&NFFL+&NCFL) flags
OIHF     @SYMON ,               Resume mapping data
```

when the @SYMOFF/ONs aren't there, the resulting mapping data becomes simply impossible for Z/XDC to interpret usefully, so a **FORMAT** command's display of this instruction winds up being simply awful!

But with the @SYMOFF and @SYMON added, the mapping data generated by the ORG and the DCs is removed, allowing Z/XDC to correctly interpret and display the modified OIFH instruction.

Note, the name fields shown on the @SYMOFF/ON macro calls are present only to give the person writing or studying the macro something useful to look at. The @SYMOFF/ON macros themselves just ignore the names.

Help MACros #Test

The **#TEST** macro contains a number of services that are very useful at assembly time. Z/XDC source and macros use #TEST extensively for various purposes.

Perhaps the most useful service is the enforcement of dependencies that are knowable at assembly time, but that are otherwise not possible or feasible to automate through other Assembler supported techniques.

Specifically, when you code:

```
    #TEST SIZE=(value1,EQ,value2)
                       NE
                       GT
                       LE
                       etc
```

A 0-length, unaligned Y-con is emitted, crafted such that an assembly syntax error occurs when the specified relation is violated. [That ought to get your attention!]

#TEST SIZE= can be used to vcheck and enforce dependencies at **assembly**

time. See **#DIE** for detecting dependency violations and illogical conditions at run time.

The #TEST macro source contains comprehensive usage and syntax commentary that you need to review before using this macro.

Help MACros #XDCHook

The **#XDCHOOK** macro generates a Static Hook within a user's program. Static Hooks are used to create debugging sessions on the fly, without requiring any additional prep. For more information, see:
- HELP HOOKS STATIC
- HELP HOOKS STATIC #XDCHOOK

The #XDCHOOK macro source contains comprehensive usage and syntax commentary that you need to review before using this macro.

Help MACros #XDCLoc8

The **#XDCLOC8** macro can be used to locate a common storage copy of the **XDC** load module without having to use z/OS's LOAD service. This can be used in SVC-unfriendly execution environments. This is called Z/XDC's **Quick Locate** support.

Note that the cblocks this macro needs for chaining through to find XDC are all located in page-fixed storage; therefore, #XDCLOC8 can also be used even when the CPU is disabled!

The aforementioned cblocks that #XDCLOC8 needs are:
- The CVT (mapped by #CVT)
- The ECVT (mapped by #ECVT)
- IBM's Customer Anchor Table (partially mapped by #CUSTABL)
- Our CSW Anchor Table (partially mapped by #DBCANCH)

These maps are all optional, but if you want to use them, their macros are available in the **XDCMACS** library.

When an assembly includes maps for the above cblocks, #XDCLOC8 will use the appropriate field names in the logic that it emits. If any or all of the mappings are not available, then #XDCLOC8 will instead use magic numbers where it has to.

We expect that #XDCLOC8 will be used in **SRB** routines, **locked** routines, **disabled** code, **FRR** protected routines, and the like. But in fact, it can be used **anywhere** when you want to avoid using z/OS's LOAD macro.

In addition to finding the XDC load module, the #XDCLOC8 macro supports locating the root load module for up to one other clone. (I call this root module the clone's **xxx** load module.) But #XDCLOC8 can provide that support only so long as that **xxx** module resides in common storage. For more information, see **HELP XDCCLONES.**

Presumedly, **xxx** would be a beta instance of Z/XDC.

The #XDCLOC8 macro source contains comprehensive usage and syntax commentary that you need to review before using this macro.

The current status of Quick Locate's support can be displayed by the **LIST XDC** command.

Help MAPs

Z/XDC can load and then use the following kinds of symbol tables ("maps").

- **Module Maps:** (Also known as linkedit maps or Binder maps) These are tables of such bind time symbols as class names, csect names, common block names, part names, and entry names (names appearing on Assembler **ENTRY** statements). Module maps show where in a load module or program object the classes, the csects, and such are located.

- **Assembler Csect and Dsect Maps:** These are tables of such assembly time symbols as statement labels and data field names. These maps show where, within a given csect or dsect, the named statements or fields are located. Z/XDC supports several flavors of csect and dsect maps:

  - **Assembler Symbol Level Maps:** These are limited maps that show information only about a program's or control block's statement labels. They do not show source statement images such as commentary and other Assembler statements.

    Symbol level maps are built from SYM data that the Assembler produces (and the Binder preserves), but only when **PARM=TEST** is specified **both** for the Assembler and the Binder.

  - **Assembler Source Level Maps:** These are more comprehensive maps that show, not only statement labels, but also a program's or control block's original source image statements, including commentary, macro expansions, etc.

    Source level maps are built from ADATA that the Assembler produces either either when **PARM=ADATA** or **PARM='GOFF(ADATA)'** is specified for the Assembler.

- **XL C/C++ and Metal C Program Maps:** These are High Level Language (HLL) source image maps for programs written in XL C/C++ and Metal C. They show a High Level Language program's original language statements and commentary.

  These maps are built from DWARF data that the HLL compilers produce when **PARM=DEBUG** is specified.

Once a map is loaded and assigned to storage, a program's internal labels are shown by Z/XDC when displaying storage and can be used in Z/XDC commands to reference storage.

Reading Map Data

Z/XDC reads Module Maps and Assembler Symbol Level Maps (SYM data for csects and dsects) from load modules or program objects residing in either PDS or PDSE type libraries. Z/XDC supports all program management levels of load modules (PM1) and program objects (PM2 through PM4, and beyond?).

In order to read Module Maps and Assembler Symbol Level Maps, Z/XDC has to OPEN a load library and scan the load module or program object "manually". When the load module resides in an old-format PDS, Z/XDC uses BSAM to read the load module records directly. When the load module or program object resides in a PDSE, Z/XDC has to use the IBM Binder's API to scan the module/object and extract the necessary information on Z/XDC's behalf.

Z/XDC builds Assembler Source Level Maps from ADATA. ADATA can reside:
- Within ADATA classes within a program object,
- Within a GOFF-type object code output file written by the Assembler to //SYSLIN,
- Or in its own RECFM=VB sequential file written by the Assembler to //SYSADATA.

Of these choices, we strongly recommend the //SYSADATA side file over the other two choices.

Z/XDC builds XL C/C++ and Metal C Program Maps from a combination of:
- DWARF data produced by the compiler,
- You own XL C/C++ and Metal C source code libraries,
- And ESD and other information from the program object itself.

Privately loaded Load Modules

A **Privately Loaded** load module is one that has been read into storage in such a way that its location is not known to z/OS. When this is the case, Z/XDC has no way to automagically acquire knowledge of such a module, leaving the user with no way display information about it. The module cannot, for example...
- Show up in **LIST PGMS** displays,
- Show up in the header lines of **FORMAT** and **WHERE** commands,
- Be displayed by the **LIST LKEDMAP** command,

However, using maps, Z/XDC provides a couple of ways by which you can notify Z/XDC where such a module exists. Then Z/XDC incorporates that knowledge and uses it in all the ways it normally would for any System loaded module. You can find detailed information in **HELP MAPS PRIVATELYLOADED.**

Modules Loaded from the USS File System

Currently, Z/XDC has no way to load ESD maps for modules that have been loaded in from Unix System Service's File System (HFS or ZFS). However, if you have copies of such modules also located in a classic PDS or PDSE, you can use Z/XDC's **Privately Loaded** module mapping support to map such modules.

AUTOMAPing

When **AUTOMAPing** is turned on, Z/XDC will automatically load Module Maps (as well as Csect/Program Maps when available) whenever Z/XDC first detects that execution has reached a module and csect for the first time. This detection occurs whenever Z/XDC

receives control (due to a breakpoint or ABEND). At such times, Z/XDC checks both the error level and retry level PSWs to see what is currently executing. If maps for those locations have not yet been loaded, then they are loaded now.

Generally, AUTOMAP operates silently. For the most part it does not issue error messages when problems occur, so if AUTOMAPing does not occur when you expect it to occur, try a **MAP PSWE?** command to find out what the problem is.

For more information, see **HELP MAPS AUTOMAPPING.**

Licensed Features and Mapping

The **asm/XDC Feature** must be Licensed, in order to use either the MAP command or the DMAP command for building csect and dsect maps from ADATA or SYM data.

The **c/XDC Feature** must be Licensed, in order to use the MAP command for building High Level Language program maps from DWARF data.

Note, the DMAP command cannot be used to build XL C/C++ and Metal C program dsects because XL C/C++ and Metal C programs do not use dsects. Their data areas are displayed via the **LIST VARIABLES** command, not the FORMAT command.

The ability to build Module Maps and Dsect Maps from a load module's (or program object's) ESD data is a part of **base/XDC,** so no special licensing is needed for using the MAP and DMAP commands for loading such maps.

For more information about Licensed Features, see **HELP SUPPORT FEATURESANDCAPS.**

Module Maps

Module Maps are built from ESD information found within nearly all load modules and program objects. So, except for load modules built with the NE attribute, a Module Map can be loaded for any load module and any program object.

Module Maps may or may not be flagged as being case-sensitive. If the map contains mixed-case names, then it will be automatically flagged as being case-sensitive; otherwise, it will not. For more information about the consequences of this, see **HELP ADDRESSING PARSERS ASM MIXEDCASE.**

Support for loading Module Maps is included in base/XDC.

Assembler Maps

Assembly time information (Csect Maps, and Dsect Maps) can be obtained by Z/XDC either from SYM data or ADATA:

- SYM data is a limited source of information that the Assembler creates (and the Binder preserves) when **PARM=TEST** is specified for both the Assembler and the Binder. When present, SYM data is always found within load modules or program objects (never in side files). Z/XDC can build Symbol Level Maps from SYM data.

- ADATA is a more comprehensive (and much more voluminous) source of information that the Assembler creates when either **PARM=ADATA** or **PARM='GOFF(ADATA)'** is specified for the Assembler. ADATA can be contained within program objects, or it can be located in side files that are entirely separate from program objects. (ADATA can never be contained within load modules.) Z/XDC can build Source Level Maps from ADATA.

Note, the Z/XDC product includes stand-alone utilities that can be used to drastically reduce both SYM data and ADATA side files by editing out unneeded and redundant data:

- For SYM data, the utility is name **XDCSYMED** and is documented at **HELP MAPS SYM EXCESSSYMDATA.**

- For ADATA, the utility is name **XDCADATA** and is documented at **HELP MAPS ADATA EXCESSADATA.**

Source Level Maps are created by Z/XDC from ADATA side files produced either by IBM's High Level Assembler or by Tachyon Software's Cross Assembler, or by Dignus' Systems/ASM.

Z/XDC can read ADATA from the following sources:

- Any file created by IBM's High Level Assembler via its //SYSADATA DD card. (These are called **ADATA side files.)**

- Any conversion of an ADATA side file to any RECFM and LRECL.

- Any ADATA file created either by Tachyon Software's Cross Assembler or by Dignus' Systems/ASM. The ADATA file can be uploaded from the user's PC into any sequential file having any reasonable RECFM and LRECL.

- From GOFF format object files created either by IBM's High Level Assembler or by Tachyon Software's Cross Assembler.

- From ADATAnnnn classes within a program object.

When reading ADATA from an ADATA file, Z/XDC determines ADATA record lengths and boundaries without regard to the file's RECFM and LRECL. Z/XDC regards the ADATA records as being streamed into the logical records without padding and without regard to record boundaries. This approach both is compatible with the way in which IBM's High Level Assembler produces ADATA, and permits Tachyon Software customers and Dignus customers to upload ADATA from the PC without having to be concerned with the structure of the mainframe file into which the ADATA is being uploaded.

Z/XDC can read ADATA from either ADATA side files, GOFF files, or program objects. When ADATA is to be obtained from ADATA or GOFF files, the file has to be made known to Z/XDC via the **SET MAPLIBS** command. This command can be used to build one or more named lists of datasets, establish one of those lists as being active, and save the extra lists in the session profile, for future use.

Assembler maps are **never** flagged as being case-sensitive. This is because even though the Assembler supports mixed-case names, it treats those names in a case-insensitive way: When two names are identical except for case, the Assembler considers them to be the exact same name. For more information about the consequences of case-insensitivity (and the lack thereof), see **HELP ADDRESSING PARSERS ASM MIXEDCASE.**

Support for loading Assembler maps is included in the **asm/XDC** Licensed Feature. For more information about Licensed Features, see **HELP SUPPORT FEATURESANDCAPS.**

High Level Language Program Maps

Z/XDC's **c/XDC** Licensed Feature provides support for debugging programs written in any of several flavors of the C language. They are:

- LE C (from IBM)
- LE C++ (from IBM)
- Metal C (from IBM)
- Systems/C (from Dignus)

See **HELP SUPPORT FEATURESANDCAPS** for more information about Licensed Features.

Z/XDC builds XL C/C++ and Metal C Program Maps from various mapping and debugging information that is created by the IBM XL C/C++ and Metal C compiler when **PARM=DEBUG** is specified for the compilation.

When **PARM=DEBUG** is specified, IBM's compiler does the following:

- The compiler suppresses object code optimization. This is so that the generated object code will have a clear relationship to XL C/C++ and Metal C program source statements.

- The compiler writes a **DWARF** data file to ddname //SYSCDBG (which must point to a sequential FB-80 dataset or PDS(E) library member). This file contains debugging information about the program including the names of all of the program source files that were read by the program compilation process.

- The compiler causes the resulting XL C/C++ and Metal C program object to contain the name of the DWARF file dataset.

!!!WARNING!!! Be aware that the debugging information produced by the XL C compiler includes the dsnames (and path names as appropriate) both of the DWARF file and all of the XL C/C++ and Metal C program's source files! This information is **hard-coded** into both the program object and the DWARF data. This means:

- Either care must be taken to ensure that both the DWARF data file as will as all source program files not be moved, renamed or changed,
- Or if you do move or rename the DWARF and/or source files, then you will need to use Z/XDC's **LIBRARYLISTS** command to let Z/XDC know where it can find the necessary files.

For more information, see **HELP DEBUGGING C.**

MAP and DMAP Command Characteristics and Differences

When **AUTOMAP** is in effect, whenever Z/XDC receives control with program execution located in a program that has not yet been mapped, Z/XDC will automatically (and silently) load whatever Module Map and Csect/Program map it can find.

However, if:

- AUTOMAP is not set,
- Or AUTOMAP is set but you don't see the maps you expect to see,

- Or you wish to load maps of programs that are elsewhere,

Then you can load maps manually (and non-silently) by issuing one or the other of the following mapping commands:

MAP - Reads Module Maps and Csect Maps (when available) and automatically assigns them to represent the storage in which the module and csect being mapped reside. For **Privately Loaded** load modules, you will have to use the MAP command's **START=** operand to notify Z/XDC where the module resides.

DMAP - Reads Dsect Maps but does not assign them to represent storage. The user has to do that separately via a **USING** command.

The **DMAP** command can also read Module Maps and Csect Maps and treat them as if they were dsects. So the **DMAP** command provides another way to map Privately Loaded load modules that Z/XDC cannot automatically find!

Both of these commands can build their maps from SYM data or ADATA (or DWARF data in the case of the MAP command). Thus, both commands can build either Symbol Level Maps or Source Level Maps.

There is a subtle but important difference in the way that the DMAP command operands (vs. the MAP command) refer to the maps that it reads:

- From Z/XDC's point of view, the first operand of the MAP command does **not** specify a load module and csect name. Instead, the first operand is processed as an arbitrary address expression, which might happen to be the name of a load module and a csect, but which might just as easily be something else such as "R15?" or "PSW?".

  In any case, for the MAP command, the first operand is first resolved to a specific location in virtual storage. Z/XDC then examines system control blocks (CDEs on the job pack and link pack queues and LPDEs in the System's PLPA directory and APEs for modules within CICS address spaces) to determine the name of the load module whose maps are to be read. The significance of this is three-fold:

  - First, the MAP command operand that Z/XDC uses to figure out what maps are desired can be **any** address expression that resolves to **any** location falling within the module (and csect) to be mapped. This might be the name of the module, or it might be a location within the module pointed to by a register or the current PSW, or it might be any other address expression that resolves appropriately.

  - Second, the load module whose module and csect maps are to be read must itself already be loaded into storage and described by a standard CDE or LPDE located on a standard system queue (or by an APE located within a CICS aspace).

    Note, this restriction is violated in the following circumstances:
    - Modules brought into storage by **directed LOAD** forms of the LOAD macro.
    - Modules loaded into common storage via the LOAD macro's **GLOBAL=YES** operand. (A CDE is **not** automatically added to the System's Dynamic Link Pack Queue.)

    See **HELP MAPS PRIVATELYLOADED** for what you need to do to map modules such as these.

  - Third, once Z/XDC has read the Module Map (and Csect Map or Source Level Map

if available), it then automatically knows where in storage those maps are to be located.

- On the other hand the first operand of the loading form of the **DMAP command** is **not** an address expression. It is a PDS(E) member name (usually a load module name) followed by a dot followed by a dsect or csect name. The implications of this are:

  - First, the load module containing the desired Dsect Map or Csect Map need not exist in storage. It only needs to reside in any accessible load library on disk, preferably (but not necessarily) within the Home Address Space's current search order.

  - Second, once Z/XDC has read the Dsect Map or Csect Map, it does **not** automatically know where in storage the map should be placed. It is not possible (for example) for Z/XDC to have intrinsic knowledge of which instance of an arbitrary control block a given Dsect Map should represent.

    Instead, the user must then issue Z/XDC's **USING** command to assign the map to the desired location.

  - Third, The **DMAP** command can be used to read, not only Dsect Maps, but also Csect Maps and even Module Maps **as if they were dsects.** This simply means that Z/XDC allows the user to have full control over the placement in storage of such maps. Thus, if a load module resides in storage but is **not** described by System or CICS control blocks, then the user still is able to map such a module using the FIND, DMAP, and USING commands instead of the MAP command. See **HELP MAPS CSECTSASDSECTS** for more information.

See also **HELP MAPS PRIVATELYLOADED** for additional information about mapping Privately Loaded load modules and making their existence known to Z/XDC.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

AUTOMAPPING - All about AUTOMAPing and its controls.
BINDERMAPS - All about Module Maps.
CSECTMAPS - All about Csect Maps.
DSECTMAPS - All about Dsect Maps.
CMAPS - All about XL C/C++ and Metal C Source Image Maps.
CSECTSASDSECTS - Loading and manipulating Module Maps and Csect Maps as if they were Dsects Maps.
PRIVATELYLOADED - Making Z/XDC aware of and then map load modules that have been Privately Loaded and, therefore, not described by system control blocks,
SYM - All about Symbol Level Maps built from SYM data.
ADATA - All about Source Level Maps built from ADATA.
DWARF - All about Source Level Maps built from DWARF data.
FLOATING - Assigning fixed or floating base addresses to Dsect Maps.
AUTOCLONING - Generating multiple maps to represent a table or chain of control

blocks.
XDCMAPS - Over 2800 preassembled IBM control block Dsect Maps available.
CICSMAPS - Hundreds of preassembled CICS control block Dsect Maps available.

Related information can be found by typing:
HELP COMMANDS DMAP
HELP COMMANDS MAP
HELP COMMANDS LIST LKEDMAP
HELP COMMANDS LIST MAPS
HELP COMMANDS USING
HELP DEBUGGING C
HELP SUPPORT FEATURESANDCAPS

Help MAPs AUTOMapping

Z/XDC can automatically load module, csect and source image maps whenever it detects that execution has reached a new module or csect for the first time. This detection occurs whenever Z/XDC receives control (due to a breakpoint or ABEND). At such times, Z/XDC checks both the error level and retry level PSWs to see what is currently executing. If maps for those locations have not yet been loaded, then they are loaded now.

Previously, if maps were wanted, the **MAP** command always had to be used to load them. Now however, whenever Z/XDC receives control, it will check the modules into which the error level and retry level PSWs point. If those locations have not yet been mapped, and if Z/XDC has not previously tried to map them, then it will do so now.

- This service can be turned on or off, by the **SET AUTOMAP** command.
- Its current setting is displayed by the **LIST AUTOMAP** command.
- This setting is saved in your Session Profile by the **PROFILE SAVE** command.
- And it can also be displayed, set and saved by the **Profile Menuing System.**

For more information, see HELP MAPS.

Help MAPs Bindermaps

A Binder map (also known as a linkedit map or module map) communicates to Z/XDC the locations of a load module's or program object's classes, control sections, parts, external labels, etc. Z/XDC requires Binder maps in order for it to know which part of a module is contained within what csect.

Binder maps can be loaded:
- Either automatically through **AUTOMAP** processing,
- Or manually by use of the **MAP** command.
- Or as dsect maps by use of the **DMAP** command.

Z/XDC builds Binder maps from a module's ESD (External Symbol Dictionary) information. For load modules stored within PDS's, Z/XDC uses BSAM to read the ESD

records directly. For load modules and program objects stored within PDSE's, Z/XDC uses the Binder's API to obtain the necessary ESD information. Z/XDC understands all program management levels of load modules (PM1) and program objects (PM2 through PM5, and beyond?).

Z/XDC's ability to build Binder maps is a part of **base/XDC.** No Licensed Feature is needed. See **HELP SUPPORT FEATURESANDCAPS** for more information.

Binder maps may (or may not) be case-sensitive. If the map contains any labels whose names contain lowercase letters, then Z/XDC flags that map as being case-sensitive. When a map is case-sensitive, then commands that reference the labels within the map must exactly match the case of those labels. See **HELP ADDRESSING PARSERS ASM MIXEDCASE** for more information.

Binder maps provide Z/XDC with knowledge of a load module's or program object's internal structure. Having this knowledge is a prerequisite for the following:

- The **SET QUALIFIER** command cannot be issued for a module until a Binder map has been read. This is because this command wants to define both a default module and a default csect. But a Binder map is necessary for Z/XDC to know which csects are contained within a load module or program object.

- A csect map cannot be loaded by the **MAP** command until the Binder map for the module containing that csect has been loaded. However, a **MAP** command issued for loading a csect map will automatically load a Binder map first whenever necessary.

  Note, a csect map **can** be loaded by the **DMAP** command independently of the Binder map, but then a **USING** command will have to be issued to assign that map to a suitable storage location. (A csect map loaded by the **MAP** command will automatically be assigned to the correct storage location.)

Mapping Load Modules and Program Objects that Z/XDC Cannot Find

Usually, whenever a load module or program object is present in storage, Z/XDC will automatically know where in storage that module is located. This is because Z/XDC knows how to scan Job Pack Queues, Link Pack Queues, the Pageable Link Pack Area (PLPA) and the like, to find module descriptors (CDEs and LPDEs). Z/XDC also knows how to scan **CICS** control blocks to find modules loaded into CICS address spaces.

Sometimes however, the storage locations of load modules and program objects is not recorded in standard system control blocks. Examples:
- Directed loads of modules into preallocated storage.
- Modules loaded into common storage for the private use of particular address spaces.

Z/XDC refers to such modules as **Privately Loaded** load modules (or program objects).

When this happens, a normal **MAP** command cannot be used to load the maps of such modules. However, if you, the programmer, have a way of knowing where such a module resides, there are two ways you can load the map anyway:

- The **MAP** command has a **START=** operand that you can use to explicitly tell Z/XDC where your Privately Loaded load module resides.

- Or you can use the **DMAP** command to read the module's map as if it were a dsect

map. Then you can use the **USING** command to assign that map to the correct location in storage. (See **HELP MAPS CSECTSASDSECTS** for details.)

You can use either of these methods to build and properly place a Binder map to represent a Privately Loaded load module or program object. **In addition,** knowledge of the existence of the module is automatically made available to the rest of Z/XDC, thereby allowing you to reference and display the module just as freely as you can for any other module in the System. Examples:

- The module will show up in **LIST PGMS** displays,
- It will show up in the header lines of **FORMAT** and **WHERE** commands,
- It will be displayable by the **LIST LKEDMAP** command,
- If you are licensed to use c/XDC, you can load source image maps for those Privately Loaded modules identified via the **MAP** command. (c/XDC cannot work with Privately Loaded modules identified via the **DMAP/USING** commands),
- etc.

For more information, please read **HELP MAPS PRIVATELYLOADED.**

Characteristics of Program Object Maps

Z/XDC’s support for Binder maps of program objects has the following characteristics:

- Loaded classes and control sections are understood. After a map has been loaded, then the following syntax can be user to reference a class, a section, or an external label:
  ```
  modulename.classname
  modulename.csectname
  modulename.labelname
  ```
  In other words, the fact that sections occur **within** classes is of no special significance to Z/XDC.

- When a section (a control section, for example) contributes content to more than one loadable class, one result is that the section becomes fragmented within the program object; that is, the section appears in multiple pieces scattered throughout the object. So, in order to give each fragment a unique name, the **MAP/DMAP** commands have to construct names that consist of the section’s name appended by the name of the class within which the fragment resides. Example: **MYCSECT_B_TEXT24**

- External names longer that 64 characters will be truncated. If this truncation results in duplicate names, there will be no direct way to reference the extra names.

- When the **DMAP** command is used to load a Binder map as if it were a dsect map, and if the program object being mapped is destined to be loaded into multiple storage extents, then the **DMAP** command will create a separate dsect map for each such extent.

  Note that if a program object has deferred load classes, that those classes are assigned by the Binder to an extent that is separate from non-deferred classes. Thus, the **DMAP** command will always create a separate dsect that maps the deferred load classes.

- When a program object contains deferred load classes, the **MAP** and **DMAP** commands

behave as follows:

- **MAP COMMAND:** The map that is built will describe only those classes that are currently in storage. Classes not yet loaded into storage will not be described. If the deferred classes are later brought into storage, then the map will have to be deleted and rebuilt in order for the newly loaded classes to be described.

- **DMAP COMMAND:** One (or more) of the dsect maps that are built will describe the deferred classes.

Commands Affected by Binder Maps

The following commands either are affected by or are relevant to Binder maps:

- **MAP:** This command loads a module's map into storage and automatically assigns it to represent the storage of that module.

- **DMAP and USING:** The **DMAP** command can be used to load a module's map into storage and treat it as if it were a dsect map. The map is **not** automatically assigned to storage. Instead, the **USING** command will have to be issued to make that assignment.

- **LIST MAPS:** This command displays a list of all loaded maps (including Binder maps) and shows information about their locations in storage.

- **LIST LKEDMAP:** This command displays the internal structure of a load module or program object. The information displayed includes the sizes and locations of all classes and csects, the locations of all entry points, etc. This information is obtained from the module's map.

- **FORMAT, WHERE, and SHOW:** These commands produce formatted displays of storage. When a Binder map covers the storage being displayed, information from the map is automatically incorporated into the display. This includes the names of any classes, csects, entry points, etc. that may occur within the displayed storage, and offset information for the displayed storage.

- **DELETE MAPS:** This command can be used to purge Binder maps (as well as any other kind of map).

- **Address Expressions:** When a Binder map is present, the names of classes, csects, common blocks, parts, and entry points from within that map can be used in address expressions. See **HELP ADDRESSING PARSERS ASM** for more information.

Help MAPs CSECTMaps

A csect map communicates to Z/XDC the internal structure of XL C programs and of Assembler control sections: statement labels, data fields, source code images, commentary, and the like. When a csect map is loaded and assigned to represent storage, the information from the map is used to format displays of that storage so that you can actually see the original source code for the program contained within

that storage. Currently (October 2015), Z/XDC supports Source Image maps for Assembler and for XL C/C++ and Metal C.

- Support for loading Assembler csect maps is part of the **asm/XDC** Licensed Feature.

- Support for XL C/C++ and Metal C source image maps is part of the **c/XDC** Licensed Feature.

For more information about Licensed Features, see HELP SUPPORT FEATURESANDCAPS.

Z/XDC supports three flavors of csect maps:

- **Symbol Level Maps:** (for Assembler) These are limited maps that show information only about a program's statement labels. Symbol level maps are built from SYM data that the Assembler produces (and the Binder preserves) when **PARM=TEST** is specified both for the Assembler and the Binder. For more information, see HELP MAPS SYM.

- **Assembler Source Image Maps:** These are more comprehensive maps that show, not only statement labels, but also a program's original statements, commentary, macro expansions, etc. Assembler source image maps are built from ADATA that the Assembler produces either when **PARM=ADATA** or **PARM='GOFF(ADATA)'** is specified for the Assembler. For more information, see HELP MAPS ADATA.

- **C Source Image Maps:** (for XL C/C++ and Metal C) These are comprehensive maps that show, your C program's language statements and also provide the knowledge allowing Z/XDC to find and interpret all of your program's data (variables, arrays, structures, templates... whatever). C Source image maps are built from DWARF data that the XL C/C++ and Metal C compiler produces when **PARM=DEBUG** is specified. For more information, see HELP MAPS DWARF.

Once a csect map has been loaded and assigned to storage, a program's internal labels (for Assembler) and statement numbers (for XL C/C++ and Metal C) are shown by Z/XDC when displaying storage. These labels and numbers can be used in Z/XDC commands to reference storage.

Assembler related csect maps can be loaded by either the MAP command or DMAP command. XL C/C++ and Metal C maps can be loaded only by the MAP command.

Using the MAP Command

The **MAP** command loads both module maps and csect maps and then automatically assigns them to represent the storage in which the module and csect being mapped actually resides. The MAP command can be used to load module maps, Assembler csect maps, and XL C/C++ and Metal C program maps.

The **MAP** command is **address driven.** It's first operand is an address expression that Z/XDC has to resolve to a specific location in storage. Z/XDC then examines system queues (Job Pack Queues, Link Pack Area, CICS control blocks, etc.) to identify the module to be mapped. It then searches system libraries to find that module, builds that module's map from ESD information from the library, and finally examines that module map to determine into which csect the resolved address falls. Only then is Z/XDC finally ready to load the requested csect map. For more information, see HELP COMMANDS MAP.

Note: Even though **modulename.csectname** is a commonly used address expression for the **MAP** command, Z/XDC does **not** take the csect's name from the given address expression. Instead, it gets it from the resolution process just described. There are several reasons for this, one of which is that Z/XDC cannot really tell that the given csectname is actually a csect name until it has a chance to actually examine the module's map.

When an Assembler control section (csect) has been assembled with a nonzero starting address, Z/XDC attempts to take that offset into account when loading, referencing, and displaying csect and dsect maps of control sections. For more information, see HELP MAPS CSECTMAPS ASMOFFSETS.

Using the DMAP Command

The **DMAP** command can load almost any kind of map (module map, Assembler csect and dsect maps but not XL C/C++ and Metal C program maps). But no matter what type of map the DMAP command loads, once loaded all such maps **become dsect maps.** That is, the DMAP command loads the map but does not assign it to represent storage. That will have to be done manually by the **USING** command.

Normally, one would prefer to use the **MAP** command to load csect maps; however, the MAP command has the dependency that the module containing the csect has to be findable via standard system control block queues. Sometimes, however, this is not the case. Sometimes, a module might reside in storage but not be known to the system. When this is the case, if you have your own way of knowing where that csect is, you can use the **DMAP** command to load the csect map and then the **USING** command to assign that map to the right place in storage. For more information, see HELP MAPS CSECTSASDSECTS.

The **DMAP** command is **name driven.** It's first operand is of the form **membername.mapname** where membername usually is the name of a load module or program object, and mapname is the name of a csect map (or dsect map) to be built. For more information, see HELP COMMANDS DMAP.

Unlike the MAP command, with the **DMAP** command, the module from which the map is to be built does not have to actually exist in storage. Instead, the given membername simply must be available either in system libraries or in a given library.

Help MAPs CSECTMaps Asmoffsets

Normally, when a csect is assembled, the assembly listings show that csect as having a starting origin of zero (00000000). That, of course, has nothing to do with the control section's actual address when it eventually is loaded into storage for execution.

When multiple csects are assembled together in a single assembly, only the first control section is shown with an origin of zero. All the remaining csects show nonzero origins. These also have nothing to do with their actual addresses once loaded into storage. And in fact, the System's load routines never even see these nonzero origins.

When a **MAP** or **DMAP** command is issued to load a map for a control section, if that control section was assembled with a nonzero origin, and if that nonzero origin information is available in the mapping data (ADATA or SYM data) from which Z/XDC is building the map, then Z/XDC will take note of that information and use it to improve the storage displayed under control of that map.

Normally, a map's **zero point** is located at the start of the map. The zero point is the location that is referenced when the map's name is given without qualification or modification. Examples:

```
    FORMAT name
    FORMAT name+offset
```

Also, when Z/XDC displays show offsets within a map, those offsets are shown relative to the map's zero point.

When a map is built from a control section that has a nonzero assembly-time origin, the map is assigned a zero point that is the csect's starting address **minus** the assembly-time origin. By doing this, Z/XDC makes it possible for the user to correlate storage displays of the control section's content with displays of that same control section from his assembly listings. For more information, see **HELP ADDRESSING DOTPLUS.**

The Visible Consequences of nonzero Assembly Time Origins

The following visible changes will occur in displays produced by the **FORMAT** command (and friends) once a csect is mapped:

- The subheader messages emitted at the top of the display will explicitly show the csect's assembly origin. Example:

```
    [...] DBCCALL.CMDLOADR+0 (ASM OFFSET: +2688), DBCCALL+2688 [...]
```

- When **SET FORMAT OFFSETS** is in effect, the offsets displayed at the left will be relative to the assembly time origin (not to the csect's physical starting location). Example:

```
    TCB#10 RB#1 (DBC496W!) -------------------------------------------
    XDC ===> F DBCCALL.CMDLOADR+0
    _  00000000_000A41F0 8f --- DBCCALL.CMDLOADR+0 (ASM OFFSET: +2688)
    _    .+2688 47F0 F01C          CMDLOADR B     X'01C'(,R15)
    _    .+268C 17
    _    .+268D E6129MID C3D4C4D3 D6C1C4D9 40
    _    .+2696 F0F661F1 F761F2F1 40
    _    .+269F F0F14BF3 F3
    _    .+26A4 90EC D00C          E6129ZID STM   R14,R12,X'00C'(R13)
    _    .+26A8 18CD                        LR    R12,R13
```

In addition, the behavior of certain address expressions will change:

- When using **.+offset,** the offset will be relative to the assembly time origin (not to the csect's physical starting location). So, continuing the examples show above:

```
    FORMAT .E6129ZID
    FORMAT DBCCALL.CMDLOADR.E6129ZID
```

```
FORMAT DBCCALL.CMDLOADR+1C
FORMAT .+2688+1C
FORMAT .+26A4
```

- These will all show code starting at +X'1C' into the CMDLOADR csect.

```
FORMAT .+2678+2C
FORMAT .+1C
```

- These, however, will not. Instead, they will fail because the given initial offsets are less than the csect's assembly time origin (2688).

- When using **.+offset,** the given offset must be equal to or greater than the csect's assembly time origin.

Note that nonzero assembly origins affect only **.+offset** address expressions. No other dot-related address expressions are affected.

Availability

A control section's assembly time origin might not always be available to Z/XDC. It is not available, of course, until a **MAP** or **DMAP** command has been issued to load the map. And even once the map is loaded, assembly time origin information might still be unavailable. Again, for more information, see **HELP ADDRESSING DOTPLUS.**

Help MAPs DSectmaps

A dsect map communicates to Z/XDC the structure of (usually) control blocks and data areas. When a dsect map is loaded and assigned to represent storage, the information from the map is used to format displays of that storage so that you can actually see the data area broken out field by field, complete with field names, and original commentary.

Dsect maps pertain only to Assembler programs, not to XL C/C++ and Metal C programs. Accordingly, support for loading dsect maps is included in the **asm/XDC** Licensed Feature. For more information about Licensed Features, see HELP SUPPORT FEATURESANDCAPS.

Z/XDC supports two flavors of dsect maps:

- **Symbol Level Maps:** These are limited maps that show information only about a data area's field names. Symbol level maps are built from SYM data that the Assembler produces (and the Binder preserves) when **PARM=TEST** is specified both for the Assembler and the Binder. For more information, see HELP MAPS SYM.

- **Source Level Maps:** These are more comprehensive maps that show, not only field names, but also a data area's original source code statements, commentary, macro expansions, etc. Source level maps are built from ADATA that the Assembler produces either when **PARM=ADATA** or **PARM='GOFF(ADATA)'** is specified for the Assembler. For more information, see HELP MAPS ADATA.

Once a dsect map has been loaded and assigned to storage, the data area's or control block's internal fields are shown by Z/XDC when displaying storage and can be used

in Z/XDC commands to reference storage.

Using the DMAP Command

Dsect maps can be loaded only by the **DMAP** command. (They cannot be loaded by the MAP command.)

The **DMAP** command can load most kinds of maps: module maps, csect maps and dsect maps (but not XL C/C++ and Metal C program maps). But regardless of map type, when a map is loaded by the DMAP command, it is treated **as if it were a dsect!** That is, DMAP loads the map but does not assign it to represent storage. That will have to be done manually by the **USING** command.

The **DMAP** command is **name driven.** It's first operand is of the form **membername.mapname,** where:
- **membername** usually is the name of a load module or program object,
- And **mapname** is the name of a map that is to be loaded as a dsect.

For more information, see HELP COMMANDS DMAP.

The thing for which the dsect map is to be built does not have to actually exist in storage. Instead, the given membername simply must be available either in system libraries or in a given library.

Help MAPs CMaps

XL C/C++ and Metal C program maps are source image maps for programs written in the several flavors of IBM's XL C language. They show the program's original language statements and commentary. These maps are built from DWARF data that the XL C/C++ and Metal C compiler produces when **PARM=DEBUG** is specified. (Z/XDC does not offer support for C programs compiled using other compilers.)

Support for loading XL C/C++ and Metal C program maps is included in the **c/XDC** Licensed Feature. For more information about Licensed Features, see HELP SUPPORT FEATURESANDCAPS.

Z/XDC builds XL C/C++ and Metal C program maps from various mapping and debugging information that is created by the XL C compiler when **PARM=DEBUG** is specified for the compilation.

When **PARM=DEBUG** is specified, the XL C compiler does the following:

- The compiler suppresses object code optimization. This is so that the generated object code will have a clear relationship to XL C/C++ and Metal C program source statements.

- The compiler writes a **DWARF** data file to ddname //SYSCDBG (which must point to a sequential FB-80 dataset or PDS(E) library member). This file contains debugging information about the program, including the names of all of the program's source files that were read by the compilation process.

- The compiler causes the resulting XL C/C++ and Metal C program object to contain the name of the DWARF file dataset.

!!!WARNING!!! Be aware that the debugging information produced by the XL C compiler includes the dsnames (and path names as appropriate) both of the DWARF file and all of the XL C/C++ and Metal C program's source files! This information is **hard-coded** into both the program object and the DWARF data. This means:

- Either care must be taken to ensure that both the DWARF data file as well as all source program files not be moved, renamed or changed,
- Or if you do move or rename the DWARF and/or source files, then you will need to use Z/XDC's **LIBRARYLISTS** command to let Z/XDC know where it can find the necessary files.

For more information, see HELP DEBUGGING C.

C, C++ and Metal C programs can be mapped regardless of whether they reside in private storage or in common. Also, even Privately Loaded modules can be source mapped so long as their existence has been identified via a **MAP** command. (Privately Loaded modules identified via the DMAP/USING commands cannot be source mapped.)

Note, the DMAP command cannot be used to build XL C/C++ and Metal C program dsects because XL C/C++ and Metal C programs do not use dsects. Their data areas are displayed via the **LIST VARIABLES** command (not the FORMAT command).

Help MAPs CSECTSasdsects

Sometimes it may be useful to load a csect map and even a load module Map as if it were a dsect map; that is, it may be useful to be able to make clones of a csect map or a Module Map and to be able to move such maps around in storage (via the **USING** command) in the same way a dsect map can be moved around.

Normally, csect maps and Module Maps are loaded via the MAP command; however, Z/XDC supports a syntax whereby you can load these maps via the **DMAP** command as well. When you do this, they are treated as if they were dsect maps. For more information, see HELP COMMANDS DMAP LOAD. Read that topic, then type HELP *NEXT (F11) to read its following topic.

Note: This capability is **not** available for XL C/C++ and Metal C Source Image Maps. These maps cannot be loaded as if they were dsect maps. They can be loaded only by the **MAP** command. However, see HELP MAPS PRIVATELYLOADED For information about mapping Privately Loaded program objects.

Useful Situations

Multiple Instances: If you have multiple copies of a load module in storage, you can use the DMAP command to read that module's Module Map (and csect maps too, if desired) as if it were a dsect map. Then you can use the **cloning form** of the DMAP command to make multiple copies of that map, each having a unique name. Then you can use Z/XDC's USING command to assign each clone to the different copies of the module.

Deferred Load Classes: If you have a program object that has deferred load classes

that have not yet been brought into storage, then the only way to obtain a map of those classes is by using the DMAP command to load the program object's map as a dsect map. A separate dsect will then be created for the deferred load classes.

Deferred Breakpoints: The ADEFERRED, TDEFERRED, and HDEFERRED commands can be used to set "deferred" breakpoints or hooks into load modules and program objects that have not yet been loaded into storage. The location of the deferred breakpoint can be given relative to:

- The module's entry address.
- The module's load point.
- The starting address of any csect contained within the load module.

If you wish to set deferred breakpoints at locations that are relative to a csect, then Z/XDC must have access to the load module's Module Map; otherwise, it won't know what and where the csects are. Unfortunately, the MAP command cannot be used to read the Module Map because the MAP command can only refer to load modules that have already been loaded into storage.

The DMAP command, on the other hand, has no such requirement. It can read maps from any load module located on disk in any accessible library. Accordingly, before issuing an ADEFERRED, TDEFERRED or HDEFERRED command, you should first use a DMAP command to cause Z/XDC to read the target load module's Module Map as if it were a dsect map. Once this is done, the address expression given on the xDEFERRED command can specify locations that are relative to specific csects within the target load module.

This works because the ADEFERRED, TDEFERRED, and HDEFERRED commands, when provided with the name of a csect within a module, scan **all** available Module Maps: both those loaded by the MAP command and those loaded by the DMAP command, both with and without assigned base addresses. So it is not even necessary that the DMAP'd Module Maps have base addresses assigned to them. It is sufficient that they merely be available for scanning by the xDEFERRED commands.

Directed LOADs: When the LOAD macro is directed (via one of its **ADDRxxx=** operands) to load a module into a specific storage location, the LOAD service does not build any CDE to represent that module. This causes the existence of the module to be invisible both to the System and to Z/XDC. So if you wish to map such a a **Privately Loaded** load module or program object, a normal MAP command won't work. Instead, you will need to use one of the following alternative methods...

- The MAP command has a **START=** operand that you can use to explicitly tell Z/XDC where your Privately Loaded load module resides.

- Or you can use the **DMAP** command to read the module's map as if it were a dsect map. Then you can use the **USING** command to assign that map to the correct location in storage.

You can use either of these methods to build and properly place a Module Map to represent a Privately Loaded load module or program object. **In addition,** knowledge of the existence of the module is automatically made available to the rest of Z/XDC, thereby allowing you to reference and display the module just as freely as you can for any other module in the System. Examples:

- The module will show up in **LIST PGMS** displays,

- It will show up in the header lines of **FORMAT** and **WHERE** commands,
- It will be displayable by the **LIST LKEDMAP** command,
- etc.

For more information, please read HELP MAPS PRIVATELYLOADED.

LOADs into Common Storage: Similarly, when you use the LOAD macro's **GLOBAL=** operand to load a module directly into common storage, yes a CDE is built, but it is built only on the loading address space's Job Pack Queue, not the System's Dynamic Link Pack Area. Thus, the module can be found by Z/XDC only when it is running in the LOAD'ing address space. It is invisible from all other aspaces.

So to map such Privately Loaded load modules, you will need to use one of the alternative methods discussed above for **Directed Loads.**

Example:

DMAP TRANSACT.
This loads a Module Map for the load module named TRANSACT; however, that map is not assigned to storage. A **LIST MAPS** command will show that the map is present but inactive.

USING TRANSACT.CALCUL8 R10?
Assuming that R10 points to the start of the control section named CALCUL8, this command positions the TRACSACT map such that the CALCUL8 label within that map falls at the address pointed to by R10.

DMAP TRANSACT.CALCUL8
This loads the csect map for the CALCUL8 control section.

USING CALCUL8 TRANSACT.CALCUL8
This command assigns the map named CALCUL8 to represent the storage starting at the location of the CALCUL8 label within the TRANSACT map.

Notice that you can use field names from the csect map to reference storage, and storage displays will be formatted under control of the map. Read on for discussions and examples of this.

Referencing Field Names

The only thing that might be confusing is that under this scheme, field names (names within csects) are qualified a bit differently.

When the normal MAP command can be used, the two maps (the Module Map and the csect map) are automatically integrated with each other so that field names become the third part of three-part names.

Further, the SET QUALIFIER command can be used to set defaults for the first two parts of such three-part names. Examples:

```
FORMAT TRANSACT.CALCUL8.TOPLOOP

***or***
SET QUALIFIER TRANSACT.CALCUL8
FORMAT .TOPLOOP
```

However, when the Module Map and the csect map are loaded separately via the **DMAP** command, this automatic integration of the two maps does not occur, and so three-part names do not occur. Therefore:

- The names of all of your csects and entry points exist as 2nd-level names within your Module Map. Examples:
  ```
  FORMAT TRANSACT.CALCUL8
  FORMAT TRANSACT.ANOTHERC
  FORMAT TRANSACT.ENTRYPNT
  ```

- The names of your csect's **instructions** and **fields** are 2nd-level names existing within your csect map. Examples:
  ```
  FORMAT CALCUL8.MYDCB
  FORMAT CALCUL8.TOPLOOP
  ```

- The name of your csect resides in **two** places, as a 2nd level name within the Module Map, and as a 1st level name within the csect map. Example:
  ```
  FORMAT TRANSACT.CALCUL8
  FORMAT CALCUL8
  ```

- Since any name within any dsect map can be referenced unqualified (i.e. without giving the map's name), the SET QUALIFIER command is not needed for simplifying the references. Examples:
  ```
  FORMAT .ANOTHERC (from the TRANSACT map)
  FORMAT .ENTRYPNT (from the TRANSACT map)
  FORMAT .MYDCB    (from the CALCUL8 map)
  FORMAT .TOPLOOP  (from the CALCUL8 map)
  ```

- Note, one the consequences of all this is that both of the following work:

  ```
  FORMAT CALCUL8
  FORMAT CALCUL8.
  ```
  These first-level-name usages work because CALCUL8 is the name of the csect map.

  ```
  FORMAT .CALCUL8
  FORMAT TRANSACT.CALCUL8
  ```
  These second-level-name usages work because CALCUL8 is also a field name within the TRANSACT map.

  ```
  FORMAT .CALCUL8.
  ```
  This usage **fails** because loading maps via the DMAP command (instead of via the MAP command) avoids the creation of three-part names.

Help MAPs Privatelyloaded

A **Privately Loaded** load module is one that has been read into storage in such a way that its location is not described by system control blocks.

Normally, load modules and program objects are brought into storage by a System Service such as LOAD or ATTACH or by CICS or by the IPL process. Accordingly, the System will create control blocks (CDEs, LPDEs, CICS APEs) to describe them, and Z/XDC will use those control blocks to find them and incorporate them into its knowledge and displays. However, here are a couple of examples of when such control blocks are **not** created:

- Modules brought into storage by **directed LOAD** forms of the LOAD macro. A CDE is **not** created and not added to the address space's Job Pack Queue.

- Modules loaded into common storage via the the LOAD macro's **GLOBAL=YES** operand. While a CDE is placed on the address space's Job Pack Queue, one is **not** also added to Common Storage's Dynamic Link Pack Queue. So while the module's presence is visible within the address space that issued the LOAD, its presence remains invisible to all other address spaces. They can see the content of the module, but they can't find it through normal System methods.

When this is the case, Z/XDC has no way to automagically acquire knowledge of such a module, leaving the user with no way to display information about it. The module cannot, for example...

- Show up in **LIST PGMS** displays,
- Show up in the header lines of **FORMAT** and **WHERE** commands,
- Be displayed by the **LIST LKEDMAP** command,

However, using maps, Z/XDC provides a couple of ways by which you can notify Z/XDC where such a module exists. Then Z/XDC incorporates that knowledge and uses it in all the ways it normally would for any System loaded module.

- **DMAP/USING:** When a **DMAP** command is used to load a Module Map as a Dsect Map, and then a **USING** command is used to assign that map to storage, Z/XDC will consider that storage as containing a Privately Loaded load module or program object. Accordingly, Z/XDC constructs internal controls that allow that module to be referenced by other commands as a load module (or program object) and to show up in displays just like any other module in the System. For more information, see HELP MAPS PRIVATELYLOADED DMAPANDUSING.

  Limitations:
  - c/XDC cannot source-map Privately Loaded modules identified via this DMAP/USING process. Instead, you have to use the **MAP** command process described next.
  - The dsect map must have been constructed from a load module's or program object's **ESD** records.
  - The name of the map must not be longer than 8 characters.

- **MAP:** Similarly, you can do pretty much the same thing with the MAP command. Normally when loading a Module Map, the MAP command has to figure out for itself the start and end points of the module being mapped and will place the map accordingly. But for a Privately Loaded load module, you have to use the **START=** operand to let Z/XDC know where the module is, and it will place the map accordingly. Then the module can be referenced by **all** Z/XDC commands **including** c/XDC's source mapping process. For more information, see HELP MAPS PRIVATELYLOADED MAP.

Once a Privately Loaded load module has been mapped, the module's presence will be known to Z/XDC, and it will be able to use that knowledge in the same ways it does for all system loaded modules. It will...

- Show up in **LIST PGMS** displays,
- Show up in the header lines of **FORMAT** and **WHERE** commands,
- Be displayable by the **LIST LKEDMAP** command,
- Etc.

For information about mapping modules loaded by Unix System Services (USS) from a ZFS or HFS file system, see HELP MAPS PRIVATELYLOADED USSFILES.

Help MAPs Privatelyloaded Dmapandusing

It's All Just a Bunch of Names

Normally, the **DMAP** command is used to load Assembler created Dsect Maps, and then the **USING** command is used to assign those maps to represent the storage containing the corresponding control blocks or data areas.

However, all maps (Dsect Maps, Csect Maps and Module Maps) are just symbol tables. They all are just collections of names, each having their individual locations (offsets), lengths and attributes. They differ only in the kinds of things they typically represent.

Anyway, I was thinking about this one day when I realized that there was no real reason why, with a small tweak, the **DMAP** command couldn't be used to load both csect maps and module maps **as if they were dsect maps!** Well, that turned out to be a really useful insight. (If you're interested to learn more, take a look at HELP MAPS CSECTSASDSECTS.)

Using DMAP-Loaded Module Maps to Reveal Privately Loaded Load Modules

One of the more interesting things you can do with DMAP-loaded Module Maps is use them to reveal to Z/XDC the presence of Privately Loaded load modules and program objects.

For complete information about the **DMAP** command, see HELP COMMANDS DMAP, but to use the DMAP command specifically to load a Module Map as a Dsect Map, just provide the module's name followed by a period. Example: **DMAP MYMODULE.**

Then use the **USING** command to assign the map to represent the storage where the Privately Loaded load module resides. For example, if R15, say, points to your module, you would issue: **USING MYMODULE R15?**

The USING command will automatically recognize that the Dsect Map being assigned is actually a Module Map. So under the covers it builds an LPDE within its internal control blocks, and it makes that LPDE available to Z/XDC's internal CDE/LPDE search routines. This causes Z/XDC to have full knowledge of the module, and that allows Z/XDC to treat the module just as comprehensively as it does other modules in the system.

Be aware that this private LPDE will not be known to the System, but it will be known to Z/XDC.

Limitation: c/XDC cannot source map Privately Loaded modules identified via this DMAP/USING process. Instead, use the MAP command process described in the next topic.

Lying

You can use the USING command to place the Module Map pretty much anywhere. In particular, you can **lie** to your heart's content...

- Yes, you can assign the map to correctly represent the storage in which your Privately Loaded module resides.
- Or you can assign the map to represent inaccessible or non-existent storage.
- Generally, you can botch it up as much as you like.

Now, I don't know why you would do such things, but you can. Z/XDC won't stop you. It will just try to cope. But be aware, the results might be strange and confusing.

What Z/XDC will **not** allow you to do is place your ESD map so as to improperly overlap another module already existing in storage.

If you do want to place your ESD map on top of an existing module, you *must* do so **congruently.** Your ESD map's starting point must coincide with the existing module's load point. Otherwise, the **USING** command will be failed.

For a discussion of a scenario where this might actually be a useful thing to do, see HELP MAPS PRIVATELYLOADED USSFILES.

Floating the Map

Another thing you can do with DMAP-loaded Module Maps is you can **float** them just like you can any other Dsect Map. This means that the location represented by the map can automatically change when the resolution of the defining address expression changes. Example: If you said, **USING MYMODULE R15? FLOAT,** then when the contents of R15 change, the location of the map also changes instantly and automatically. For more information, see HELP MAPS FLOATING.

Deleting Privately Loaded Load Modules

The misleading answer is, you can't. The actual removing of such a module from storage is going to be up to the program that brought it in. Z/XDC's **DELETE MODULES** command can't touch it. (The DELETE command uses the System's DELETE macro, and he's the guy that's not going to work.)

But what you **can** do is delete Z/XDC's **knowledge** of the module. It's simple. all you do is:

- Either use the **DROP** command to undo the USING.
- Or use the **DELETE MAPS** command to delete the module map altogether.

And boom! The module disappears from Z/XDC's knowledge and from its displays.

An alternative way to create knowledge of a Privately Loaded load module is to use the **MAP** command. See HELP MAPS PRIVATELYLOADED MAP for details.

Help MAPs Privatelyloaded Map

Users often aren't aware of this, but generally the first operand of the **MAP** command is **not** the name of the module to be mapped. It actually is an address expression that resolves to a location in storage within the module to be mapped. Z/XDC then searches system control blocks (CDEs, LPDEs and CICS APEs) to discover the load module that contains this location.

Now it certainly is true that such an address expression can, in fact, be the name of the module to be mapped, but it also can just as easily be a PSW reference, a register reference (Example: MAP R15?) or any other address expression, just so long as the address that it resolves to is located within the load module or program object that you want to map.

Even when the address expression that you use is a load module name, Z/XDC does not use that name. Instead, it resolves that name to the storage where that module is located, and then it searches system queues to eventually discover the name of the module that way.

This process does not work for Privately Loaded load modules because, of course, such modules are not described in system control blocks, so as far as Z/XDC can determine, the address you give it is not located with any module at all.

The MAP Command's START= Operand

When you wish to load a map for a Privately Loaded load module, you can tell the **MAP** command of this by using the **START=addressexpression** operand. The **addressexpresssion** must be an expression that resolves to the Privately Loaded load module's **Load Point,** i.e. the location in storage where the module physically starts.

Often, a module's Load Point it also its Entry Point Address, but not always. Be aware of this, and be careful to provide the module's Load Point, not its Entry Point (especially when they differ).

When the **START=** operand is given, Z/XDC is alerted to the fact that this map is for a Privately Loaded load module. Therefore, its existence will not be described by any system control block. Therefore, Z/XDC can skip searching those queues.

Clearly, the address expression given via the START= operand cannot include the Privately Loaded module's name, It may start with a PSW reference. It may start with a register reference. It may start with any other reference you like, but it may not start with the module's name. This is because the module's own name will not be resolvable until **afer** the command completes, not before.

The MAP Command's First Operand (Also an Address Expression)

The MAP command's **first** operand is an address expression, but please try not to confuse this address expression with the address expression that is the value of the START= operand. I will try to be precise in the following discussion.

The first thing that is important to understand is that just because you're using the START= operand, that doesn't mean that the MAP command's first operand doesn't still have a role to play. You still have to provide it, and it still serves two important purposes...

- First, it provides the **name** of the Privately Loaded load module. (See below).

- Second, When it eventually is resolved, the resolved location will be used to figure out which Csect Map (if any) the MAP command is also supposed to load.

But remember, the first operand cannot be resolved until after the Module Map is built and assigned to the storage indicated by the START= operand. So the first operand requires some special handling, and there are constraints.

- So as just noted, MAP command processing does delay resolution of the first operand until after the Module Map has been assigned to storage.

- Then the first operand is resolved for the sole purpose of determining which Csect Map needs also to be loaded.

- The first operand address expression **must** either be or start with the name of the load module to be mapped.

- This name **must** be the name of the PDS[E] library member from which the module was loaded. The MAP command uses the name to identify the load module and the PDS[E] load library from which it came.

- The address expression may not start with a PSW reference or a register reference or anything else other than the module name as described above.

- The module name may be qualified with a csect name and/or with any other arithmetic that ends up pointing to some location within the module. Eventually this location will be used to identify and attempt to load a map for the csect that contains that location.

This address expression is **not** used to find the storage containing the Privately Loaded load module. In fact it is not even resolved until the module has already been found via the **START=** operand and after a Module Map for that module has been built. Only at that time is the expression fully resolved to determine which csect map should also be loaded.

Example

Suppose the following:
- Suppose that a module named **MYPGM** has been Privately Loaded into storage.
- Suppose that MYPGM contains a csect named **MYSECT.**
- Suppose that R12 points to MYPGM's Load Point.

Then the command **MAP MYPGM.MYSECT START=R12?** will do the following:
- It first will load MYPGM's Module Map.

- It then will notify the rest of Z/XDC of the existence and location of MYPGM.
- It finally will load MYSECT's Csect Map.

Lying

You can use the START= form of the MAP command to place the Module Map pretty much anywhere. In particular, you can **lie** with the START= operand to your heart's content...

- Yes, you can assign the map to correctly represent the storage in which your Privately Loaded module resides.
- Or you can assign the map to represent inaccessible or non-existent storage.
- Generally, you can botch it up as much as you like.

Now, I don't know why you would do such things, but you can. Z/XDC won't stop you. It will just try to cope. But be aware, the results might be strange and confusing.

What Z/XDC will **not** allow you to do is place your ESD map so as to improperly overlap another module already existing in storage.

If you do want to place your ESD map on top of an existing module, you *must* do so **congruently.** Your ESD map's starting point must coincide with the existing module's load point. Otherwise, the **MAP** command will be failed.

For a discussion of a scenario where this might actually be a useful thing to do, see HELP MAPS PRIVATELYLOADED USSFILES.

Deleting Privately Loaded Load Modules

The misleading answer is, you can't. The actual removing of such a module from storage is going to be up to the program that brought it in, Z/XDC's **DELETE MODULES** command can't touch it. (The DELETE command uses the System's DELETE macro, and he's the guy that's not going to work.)

But what you **can** do is delete Z/XDC's **knowledge** of the module. It's simple. all you do is use the **DELETE MAPS** command to delete the module map altogether. And boom! The module disappears from Z/XDC's knowledge and from its displays.

An alternative way to create knowledge of a Privately Loaded load module is to use the **DMAP** and **USING** commands. See HELP MAPS PRIVATELYLOADED DMAPANDUSING for details.

Help MAPs Privatelyloaded Ussfiles

While c/XDC does have support for loading DWARF based maps from USS File System files, that support has not yet been extended to Module maps, ADATA maps and SYM data maps. Unfortunately, if you need to map such a module, there is no easy solution, but there is a workaround.

- First, you have to copy your load modules, program objects and/or ADATA side files over to a classic PDS or PDSE libraries.

- Then by treating the USS loaded load module as if it were **Privately Loaded,** you can use the **MAP** command with the **START=** operand to load its Module Map out of the PDS[E] and place it on top of the USS loaded module.

- Then for SYM data and ADATA maps, just use the MAP and DMAP commands in their normal ways.

For details, see HELP COMMANDS MAP PRIVATELYLOADED.

Help MAPs Sym

When the **TEST** option (e.g. **PARM=TEST)** is specified for an assembly, all mainframe assemblers (both from IBM, and from Tachyon Software, and from Dignus, LLC) produce comprehensive symbol maps for all csects and dsects being assembled. These maps are written to object files in the form of **SYM** records. The maps contain entries for nearly all named objects found in the assembly. They also contain entries for all unnamed data fields (DS and DC statements). In fact about the only Assembler statements that do not generate map entries are unnamed machine instructions and EQU statements (e.g., "FLAG EQU X'80'", but see below.).

Please take note of the following special information:

- Support for loading SYM data maps is included in the the **asm/XDC** Licensed Feature. For more information about Licensed Features, see HELP SUPPORT FEATURESANDCAPS.

- Z/XDC reads SYM data from load modules, not object files. Therefore, in order for SYM data to be available to Z/XDC, you have to specify the TEST option, not only for the assembly, but also for the binding of the load module or program object containing your program. Otherwise, the Binder will discard the SYM data.

- The very old VS Assembler (IFOX00) **did** produce SYM table entries for relocatable EQUs (e.g., "BASE EQU *"). Unfortunately, though, the IFOX00 Assembler is rather obsolete, and IBM's current Assembler (the "High Level Assembler", ASMA90) does not provide EQU information in SYM data.

  This affects programmers who habitually use **EQU *** to define labels within their code. Such labels will not be available to Z/XDC from the SYM data; consequently, they will not show up in csect maps. To accommodate this failing, it is suggested that a text editor be used to change all "EQU *"s to **DS 0X's.**

- Tachyon's z/Assembler, like the ancient VS Assembler, also produces map entries for relocatable EQUs. But (unlike the VS Assembler) it is currently vendor supported, and it is compatible with IBM's High Level Assembler. (It has essentially all the same features and capabilities of the High Level Assembler.) But in order to get the EQU entries, you have to specify **TEST(XDC)** (not just TEST) among your Assembler options. For more information, see HELP TACHYON.

- Starting with Assembler H and continuing with the High Level Assembler, IBM has allowed programmers to define and use field names and statement labels that are longer than 8 characters. Unfortunately, IBM did not make the necessary changes to the structure of SYM record entries to allow them to support names longer than 8 characters. Therefore, SYM table entries for statements having long names

are created by these assemblers, but the entries' symbol name fields are left blank. Consequently, Z/XDC cannot show such long symbol names when displaying mapped storage.

- On the other hand, Tachyon's z/Assembler does produce SYM table entries for long names. If you are using z/Assembler for your assemblies, then specify the **SYMNAME(XDC)** assembly option. This will cause z/Assembler to create SYM table entries for the long names, and then you will be able to display and use long names during your Z/XDC debugging sessions.

Many of the limitations of SYM data mentioned above do not exist when Z/XDC builds its maps from ADATA. However, the management of ADATA has its own problems. For a discussion of the relative merits and demerits of SYM data vs. ADATA, see HELP MAPS ADATA PERFORMANCE.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

EXCESSSYMDATA - Managing excess SYM data when multiple assemblies containing duplicate dsect maps are linkedited together into one load module.

Help MAPs Sym Excesssymdata

Assembly time field names and statement labels are contained in **SYM** data created by the Assembler and found in both object files and load modules. SYM data is built by an Assembler when the **TEST** option is specified (e.g. **PARM=TEST).** It can then be copied by the Binder from object files to load modules when **PARM=TEST** also is specified for the linkedit. See **HELP MAPS SYM** for more information.

Another source of symbol information (plus entire source images and a whole lot more) is **ADATA.** See **HELP MAPS ADATA** for more information.

The Assembler constructs SYM data entries for a lot of stuff, including all labeled machine instructions and all DS and DC statements regardless of labeling. Also, SYM data entries are created for all statements occurring in dsects as well as csects.

Why Excess SYM Data is Created and Its Consequences

Often large programs (such as Z/XDC and such as JES2) are constructed from a large number of separately assembled control sections (csects). Typically each such assembly contains a full set of dsect maps which can run to several thousands of symbols, and many assemblies will contain the same dsect maps as many other assemblies. (This is true regardless of whether or not **PRINT OFF** is specified.) Consequently, when these separately assembled csects are brought together into one or more load modules, the load modules will wind up containing multiple, redundant copies of the dsect symbols. Thus, the final size of the load module's disk image

can become considerably larger than it needs to be.

If some method were available by which the excess and redundant symbols could be eliminated, then the size of the load module's disk image could be made considerably smaller, and the time taken by Z/XDC for searching the SYM table could be considerably reduced.

(Note, the size of the load module's storage image is unaffected by the presence or absence of SYM data, since SYM record information is always ignored by the System's LOAD, LINK, ATTACH, and XCTL services.)

Using SYMDEL/SYMUNDEL/SYMNODEL to Eliminate Excess SYM Data

Two tools are available for ridding load modules of excess symbols. One is the **XDCSYMED** utility that is distributed as part of the Z/XDC product. The other is the Tachyon Cross Assembler available from Tachyon Software. (See **HELP TACHYON.)** These two tools are described in detail in the following topics. The remainder of this topic describes the triggers that you have to add to your program source code by which these tools can determine which symbols are to be discarded and which are to be kept.

XDCSYMED and the Tachyon Cross Assembler both remove unwanted symbols by scanning for certain triggering symbols that you have to add to your Assembler source programs. These symbols are:

SYMDEL - Start deleting symbols. (nestable)
SYMNODEL - Stop deleting symbols. All nested SYMDELs are canceled.
SYMUNDEL - Cancel (unnest) a corresponding prior SYMDEL. If no SYMDELs remain nested, then stop deleting symbols.

The best way to use these **SYM Data Filtering Triggers** is to place them in the name field of DSECT statements. For example, when you have a set of symbols that you want to have deleted from the object file, place a **SYMDEL DSECT** statement ahead of them in your source code and a **SYMUNDEL DSECT** statement after them.

Then follow each SYMxxDEL DSECT statement with a suitable CSECT, DSECT, RSECT or LOCTR statement to resume the section that the SYMxxDEL statement interrupted.

If you are writing dsect mapping macros, then you could add SYM Data Filtering Triggers into the macros both before and after the dsect that the macros generate. You could add a SYMDEL DSECT statement just ahead of the dsect being generated, and a SYMUNDEL DSECT statement immediately following the dsect's last statement.

What the SYMDEL/SYMUNDEL/SYMNODEL Triggers Do

The SYMDEL, SYMUNDEL and SYMNODEL triggers are processed (by either the XDCSYMED utility or the Tachyon Cross Assembler) as follows:

- Every time a **SYMDEL** symbol is encountered, a nesting counter is incremented, and SYM data either begins to be or continues to be suppressed.

- When a **SYMUNDEL** is encountered, the nesting counter is decremented by one (but not if it already is zero). If the counter reaches zero (or remains at zero), then SYM data suppression is discontinued; otherwise, suppression continues.

- When a **SYMNODEL** is encountered, the nesting counter is forced to zero, and SYM data suppression is unconditionally discontinued.

SYMDEL/SYMUNDEL pairs can be used in macros, for example, when you want to temporarily suppress SYM data entries and then restore the prior suppression state without having to know what that state was.

SYMNODEL can be used to cancel SYM data suppression unconditionally and regardless of the SYMDEL nesting level.

The @SYMOFF and @SYMON Macros

These macros are a way to emit SYMDEL and SYMUNDEL dsects (respectively) in as easy and nondisruptive a way as possible. (There is no macro for emitting SYMNODELs.)

When @SYMOFF and @SYMON bracket a section of code, they emit the SYMDEL and SYMUNDEL triggers that will cause the XDCADATA and XDCSYMED utilities to purge all ADATA and SYM data emitted by the bracketed code. The triggers are emitted silently so as not to clutter up your assembly listings.

The macros accept almost no operands. If any are given, they are mostly ignored. (Ditto for the name field.) For complete usage details as well as several examples, see **HELP MACROS @SYMOFF/ON.**

A Note About Excess ADATA

The above excess SYM data problem also exists for ADATA, only it’s worse since ADATA is much more voluminous than SYM data. But like SYM data, excess ADATA can be controlled by proper use of SYMDEL and friends. See **HELP MAPS ADATA EXCESSADATA** for more information.

More Information

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

EXAMPLES - Examples and suggestions regarding the usage of the SYM Data Filtering Triggers.
XDCSYMED - Performing SYM Data Filtering with the XDCSYMED utility.
TACHYON - Performing SYM Data Filtering with the Tachyon Cross Assembler.

Help MAPs Sym Excesssymdata Examples

Typically, excess SYM data entries occur when a load module is created from several,

separately assembled csects. The load module accumulates all of the SYM data from all of the separate assemblies. This includes, not only the csect related SYM data, but also all SYM data for all copies of all dsects used in the assemblies. In the worst (and not uncommon) case, this could be as many as one copy of each dsect's SYM data from each assembly... all gathered together into your poor load module!

As described in detail in the prior topic **(HELP MAPS SYM EXCESSSYMDATA),** excess SYM data can be filtered by using the special symbols **SYMDEL, SYMUNDEL** and **SYMNODEL** on DSECT statements appropriately inserted into your source code. Here is an example.

A Typical Case

Typically, when a program or product is built from multiple assemblies, most assemblies will include a dsect maps definition section for both system control blocks as well as for control blocks that are private to the program. In all assemblies but one, insert a **SYMDEL DSECT** statement ahead of your collection of mapping macros, and then a **SYMUNDEL DSECT** statement afterwards, as shown below. (Notice that after the SYMUNDEL DSECT, you probably have to then resume your program's csect.) Example:

```
MYPROGRA CSECT ,
         ...
SYMDEL   DSECT ,                DELETE SYM DATA
         PRINT   NOGEN          HEY, LOOK IT UP
         IHAASCB ,              ASCB MAP
         CVT     DSECT=YES,PREFIX=YES,LIST=YES
         DCBD    DSORG=(PS,PO)  DCB MAP
         IHAPSA  ,              PSA MAP
         IKJTCB  LIST=YES       TCB MAP
         PRINT   ON,GEN,NODATA  RESUME NORMAL DISPLAY
SYMUNDEL DSECT ,                STOP DELETING SYM DATA
MYPROGRA CSECT ,                RESUME NORMAL CSECT
         ...
```

After assembling the above, you would then run the Assembler's //SYSLIN output file through the **XDCSYMED** filtering program. Then all of the SYM data generated for all of the dsect maps occurring between the SYMDEL and SYMUNDEL statements will be completely eliminated from the object file! This can lead to a substantial reduction in the DASD space required for your load module.

Please notice the following:

- Generally a SYMDEL, SYMUNDEL and SYMNODEL DSECT statement should be followed immediately by another CSECT, DSECT, RSECT or LOCTR statement so that correct dummy section and control section construction is not disrupted. You will probably have to do this yourself (as shown above following the SYMUNDEL DSECT statement).

- As many SYMDEL, SYMUNDEL and SYMNODEL DSECT statements may be used as needed. Thus, scattered groups of SYM data can be deleted easily.

- The SYMDEL, SYMUNDEL and SYMNODEL symbols can be used on any type of Assembler statement, but only DSECT statements permit repeated occurrences of a symbol, so using them on DSECT statements is most convenient.

An Alternative to SYMDEL and SYMUNDEL

@SYMOFF and **@SYMON** are a pair of macros that can be used instead of SYMDEL and SYMUNDEL (respectively) for filtering unwanted ADATA and SYM data from the Assembler's output files. They are a little easier to use than SYMDEL and SYMUNDEL, and they are designed to reduce the visual clutter in assembly listings. For usage details and examples, see **HELP MACROS @SYMOFF/ON.**

A Suggestion

As mentioned above, you should delete the excess SYM data from all assemblies but one. Let us suggest that the one assembly that you do not delete the SYM data from be a special **DOC** assembly (similar to the HASPDOC assembly used by JES2, or similar to the XDCMAPS assembly distributed with the Z/XDC product).

In other words, create a special assembly that has nothing but all of the control block maps for your program. You can then linkedit that assembly into a separate **xxxxxDOC** (for example) load module. Then Z/XDC's **DMAP** command can use that module as a source for the dsect maps for all of your program's control blocks. (Example: **DMAP xxxxxDOC.MYCBLOCK**)

Help MAPs Sym Excesssymdata Xdcsymed

XDCSYMED is a support utility that is provided as part of the Z/XDC product. Its primary purpose is to filter excess and unneeded SYM data entries by acting upon the "SYM Data Management Triggers": SYMDEL, SYMNODEL and SYMUNDEL. These triggers are described in the prior topic, HELP MAPS SYM EXCESSSYMDATA.

XDCSYMED reads object files created by an Assembler, deletes unwanted SYM data from it, performs a few other SYM data edits, and writes the revised object files to an output file for subsequent processing by a Linkage Editor or Binder. Accordingly, a good way to run XDCSYMED is to add an XDCSYMED step to your assembly JCL PROCs immediately following the assembly step.

Note, XDCSYMED is needed only if you are using an IBM written Assembler (such as the High Level Assembler) for performing your assemblies. If you are using the Tachyon Cross Assembler (from Tachyon Software), then XDCSYMED is not needed. See the next topic (HELP MAPS SYM EXCESSSYMDATA TACHYON) for more information.

XDCSYMED filters or edits only an object file's SYM data records. All other object records are copied to the output file unchanged.

XDCSYMED processes only legacy format object files. It cannot handle GOFF files. (For more current technology, see HELP MAPS ADATA EXCESSADATA XDCADATA for information about filtering ADATA side files from whatever source [including from GOFF files].)

XDCSYMED's Primary Service

XDCSYMED filters out unwanted SYM data by scanning for certain "SYM Data Management Triggers" that you have to add to your Assembler source programs. Those triggers are DSECT statements having the following name fields:

SYMDEL - Start deleting symbols. (nestable)
SYMNODEL - Stop deleting symbols. All nested SYMDELs are canceled.
SYMUNDEL - Cancel (unnest) a corresponding prior SYMDEL. If no SYMDELs remain nested, then stop deleting symbols.

See HELP MAPS SYM EXCESSSYMDATA for more information.

Additional XDCSYMED Services

In addition to the above, XDCSYMED performs the following edits:

- XDCSYMED deletes all SYM data entries representing unnamed statements having a zero length modifier (example: "DS 0X" but not "DS 0XL12").

- For csects having nonzero starting origins, it adds information to appropriate SYM data entries so that Z/XDC can know what those origins are. (This service is necessary only for the IBM written assemblers. It is not needed when the Assembler being used is the Tachyon Cross Assembler.)

XDCSYMED JCL

XDCSYMED is extremely simple to run. It uses three files: SYSIN, SYSPUNCH, and (optionally) SYSPRINT. It does not use either PARM field parameters or other input parameters.

SYSIN
This must contain an object file that was produced by an Assembler and is to be filtered by XDCSYMED. It must be a sequential dataset or a specific member of a partitioned dataset. Its DCB attributes must be:
RECFM=FB,LRECL=80,BLKSIZE=n*80
Its BLKSIZE may be any multiple of 80 that is not greater than than 32,760.

Warning: XDCSYMED only understands legacy format object files. It does not understand GOFF format object files. Accordingly, if you are going to use XDCSYMED, you must not specify or imply the XOBJECT or GOFF options to the Assembler.

SYSPUNCH
This will contain the filtered output file produced by XDCSYMED. It must be a sequential dataset or a specific member of a partitioned dataset. It may not be the same dataset as SYSIN. Its DCB attributes must be:
RECFM=FB,LRECL=80,BLKSIZE=n*80
Its BLKSIZE must be a multiple of 80.

SYSPRINT
This file is optional. If provided, then XDCSYMED will write 4 or 5 processing information messages to it. Usually, this file should be allocated to spool; however, it may be allocated to any sequential dataset or to a specific member of a partitioned dataset. Its RECFM, LRECL and BLKSIZE may be set to any reasonable values. Example:

```
        RECFM=VBA,LRECL=125,BLKSIZE=0
    If no DCB is specified, then default settings are used.
```

JCL Example

The following JCL PROC assembles a program, filters its object file, and linkedits the filtered file into a component load library.

```
//ASMCL   PROC MODULE=

//ASM     EXEC PGM=ASMA90,PARM='TEST,OBJECT'
//SYSLIB   DD  DISP=SHR,DSN=SYS1.MACLIB
//SYSUT1   DD  UNIT=VIO,SPACE=(CYL,(1,1))
//SYSTERM  DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//SYSPUNCH DD  SYSOUT=B
//SYSIN    DD  DISP=SHR,DSN=DBCOLE.ASM(&MODULE)
//SYSLIN   DD  DISP=(,PASS),UNIT=SYSDA,SPACE=(TRK,(5,5),RLSE),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=0),DSN=&&OBJECT

//SYMEDIT EXEC PGM=XDCSYMED
//SYSPRINT DD  SYSOUT=*
//SYSIN    DD  DISP=(OLD,DELETE),DSN=&&OBJECT
//SYSPUNCH DD  DISP=(,PASS),UNIT=SYSDA,SPACE=(TRK,(5,5),RLSE ,
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=0),DSN=&&FILTERED

//LKED    EXEC PGM=IEWL,PARM='TEST,NCAL'
//SYSPRINT DD  SYSOUT=*
//SYSUT1   DD  UNIT=VIO,SPACE=(CYL,(1,1))
//SYSLMOD  DD  DISP=SHR,DSN=DBCOLE.LKED(&MODULE)
//SYSLIN   DD  DISP=(OLD,DELETE),DSN=&&FILTERED

//        PEND
```

Notice that PARM=TEST has been specified for both the assembly step **and** the linkedit step.

Help MAPs Sym Excesssymdata Tachyon

Tachyon Software makes an Assembler named **z/Assembler.** This is a great(!) Assembler. It runs on your PC (DOS, Windows, OS/2, or UNIX) instead of the mainframe, but it produces object code that is Binder ready for the mainframe. All you have to do is upload the Assembler's output to your mainframe, run it through the Binder, then execute it. For more information, see HELP TACHYON.

When the Tachyon Cross Assembler is used for assembling your programs, in can automatically look for and process the SYMDEL, SYMNODEL, and SYMUNDEL "SYM Data Management Triggers" that XDCSYMED processes. The Tachyon Assembler will perform the very same symbol deletions that XDCSYMED performs. In fact, the Tachyon Assembler also will perform all the other editing and cleanup functions that XDCSYMED does. Therefore, when the Tachyon Assembler is used, the XDCSYMED utility is not needed at all. (See HELP MAPS SYM EXCESSSYMDATA.)

To enable these functions, all you have to do is specify the TEST(XDC) Assembler option when you run the Tachyon Cross Assembler. See Tachyon's documentation for more information.

Help MAPs AData

ADATA is a file of data that contains comprehensive (one might even say excessive) information concerning Assembler programs. Both the High Level Assembler (ASMA90), and the Tachyon z/Assembler and Dignus' Systems/ASM have options for producing ADATA.

Support for loading ADATA maps is included in our **asm/XDC Licensed Feature.** For more information about Licensed Features, see HELP SUPPORT FEATURESANDCAPS.

What ADATA Contains

Z/XDC can use ADATA for producing source level maps of programs and control blocks located in storage. These maps contain the following information:

- The names, locations, lengths, and attributes of **all** statement labels and field labels defined in the Assembler program. This includes **long** labels (not available from SYM data).

- The names and locations of all Assembler **EQU** symbols, both relocatable and **non-relocatable** (also not available from SYM data).

- Exact images of all of a program's source statements **(including** all source statements generated by dsect mapping macros regardless of whether or not displays of those statements have been suppressed in the assembly listing!)

- All macro calls as well as indications of which statements following a macro call are macro expansion statements.

- All comment statements.

- For machine instructions, the original object code generated by those instructions.

- All cross reference tables produced during an assembly.

- Way too much other stuff!

Z/XDC's Handling of Both Relocatable and Non-Relocatable EQUs

The following are some notes regarding Z/XDC's mapping and handling of Assembler EQUs as presented in ADATA:

  - Assembler EQUs can be either relocatable or non-relocatable.

  - **Relocatable** EQUs are treated like any other label: They resolve to the

addresses to which they are assigned. In fact, they are correctly resolved by Z/XDC even when they are assigned to a csect or dsect different from that within which they appear!

- **Non-Relocatable** EQUs, of course, do not resolve to any address. They represent numbers or strings, not locations.

  Nevertheless, they can still be used within address expressions as if they were relocatable references to the place, within an ADATA map, at which they are defined. That way it is easy to see what an EQU's definition is.

  This is particularly useful for EQUs that define flag names. Typically, such EQUs occur in source code immediately following the DS statement for the flag byte to which they apply. So when you use such an equate in an address expression, you wind up referencing the location of the flag byte (not just the value of the flag).

- Assembler EQUs are not available from SYM data maps (only from ADATA maps).

- For more information, see HELP ADDRESSING PARSERS ASM ASMEQUS.

Z/XDC can make use of all this information to produce csect and dsect maps that are exceedingly informative. When ADATA is present, storage displayed by the FORMAT and WHERE commands can show the original source code images and commentary for the programs and control blocks being displayed. In fact, Z/XDC will even recognize when a program's machine instructions have been **changed** and will display **hilighted warnings** about it!

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

CREATING - How to create ADATA and where Z/XDC can find it.
MAPLIBS - Providing access to ADATA residing outside of a program object.
SEARCHORDER - The process by which Z/XDC searches for ADATA and SYM data.
CACHING - Information about ADATA caches created by Z/XDC.
PERFORMANCE - Performance issues regarding ADATA.
EXCESSADATA - Mitigating excessive ADATA.
MACROS - Showing or suppressing macro expansions.
TIPS - Here's a tip or two regarding ADATA.

For information about Z/XDC commands pertaining to ADATA, please select any of the the following:

HELP COMMANDS MAP
HELP COMMANDS DMAP
HELP COMMANDS SET MAPLIBS
HELP COMMANDS LIST MAPLIBS
HELP COMMANDS DELETE MAPLIBS

```
HELP COMMANDS LIST CACHE
HELP COMMANDS DELETE CACHE
HELP COMMANDS SET FORMAT
HELP ADDRESSING PARSERS ASM
```

Help MAPs AData CReating

The High Level Assembler can generate and write ADATA either:
- As part of GOFF object file output (written to the //SYSLIN ddname)
- Or as an entirely separate ADATA file (written to the //SYSADATA ddname)
- Or as a **reduced size** ADATA file (written either to a GOFF file or to a //SYSADATA ddname or to a **//xxxADATA** ddname)
- Or as all three simultaneously.

These choices are controlled by the following Assembler options:

PARM=ADATA
This causes the Assembler to generate and write ADATA to the SYSADATA ddname.

PARM='GOFF(ADATA)'
This causes the Assembler to do the following:
- Write its object output to the SYSLIN ddname using the extended GOFF format ("Generalized Object File Format").
- Embed ADATA into the object output.

Eventually when the object file is processed by the Binder, the ADATA will be included into the resulting program object as a series of nonloadable ADATAnnnn classes.

PARM='ADATA,GOFF(ADATA)'
This causes the Assembler to write its ADATA both to the SYSADATA ddname and to the object output file (SYSLIN).

PARM='EXIT(ADEXIT(xxxADATA))'
This causes the Assembler to invoke an ADATA processing exit routine (that we provide) that filters the ADATA and optionally:
- Writes it out to an xxxADATA ddname,
- Or returns it to the Assembler so that it can be written out (by the Assembler) either to SYSADATA or to a GOFF file (or to both).

For details, see HELP MAPS ADATA EXCESSADATA XDCADATA.

Consequently, ADATA can occur in three different places:
- By itself in a sequential file (or PDS/PDSE member).
- Embedded within a GOFF format object file.
- Embedded within a program object.

Z/XDC is capable of reading ADATA from any of these places.

When ADATA is embedded within a program object, Z/XDC's MAP and DMAP commands will find it and read it using the Binder's API.

When ADATA is located within its own file or within a GOFF object file, then the MAP and DMAP commands can read it, but first the **SET MAPLIBS** command has to be used so that the MAP/DMAP commands will know what dataset and libraries it should search when looking for the ADATA.

When a large program containing many csects is built from multiple assemblies, some

programmers will set up the JCL such that the ADATA output for each assembly will be written to separate members of a partitioned dataset, while other programmers will prefer to append all the ADATA together into a single sequential dataset (or PDS(E) member). Z/XDC can accept and process ADATA from either structure: neatly separated by csect into PDS(E) members or piled altogether into a single file. See HELP *NEXT (F11) (i.e. HELP MAPS ADATA MAPLIBS) for more information.

WARNING

The volume of ADATA can be **massive!** For example, the ADATA files generated by a full assembly of Z/XDC itself (some 200+ assemblies), if I were to generate it all, would require around **55 thousand** tracks on a 3390! For me, that's out of the question.

Fortunately, there is a solution. One of the utility programs that comes with Z/XDC is named **XDCADATA.** It is a filtering program that can be used to reduce the size of ADATA files by 50% to 90% or so. For details, see HELP MAPS ADATA EXCESSADATA XDCADATA.

Help MAPs AData MAPlibs

The **SET MAPLIBS** command can be used to build one or more lists of one or more sequential datasets and/or partitioned datasets (libraries) that the MAP and DMAP commands will search while building source image csect and dsect maps. The datasets within an active MAPLIBS list can have the following characteristics:

- Each dataset can be one of the following:
  - A sequential dataset.
  - A specific member of a partitioned dataset.
  - An entire partitioned dataset (either PDS or PDSE).

- Each dataset or PDS(E) member can contain either raw ADATA or GOFF formatted object data containing embedded ADATA.

- There is no requirement that all datasets and/or members in a list be of the same type.

- The DCB attributes of the various datasets in a MAPLIBS list can be anything reasonable, including anything supported by the High Level Assembler.

- There is no requirement that all datasets have matching DCB attributes.

- For ADATA side files (but not for GOFF files), the ADATA can be streamed into the dataset without padding and without regard for record boundaries. In other words, ADATA records do not need to start or end at logical record boundaries. Thus, when an ADATA record ends within a logical record, the next ADATA record **must** start with the next following byte.

- This capability allows Z/XDC to support ADATA that is created on a PC (such as by Tachyon's Cross Assembler) and then is stream uploaded into an arbitrary mainframe dataset with arbitrary RECFM and LRECL.

- Yet this capability remains compatible with ADATA as produced by IBM's High Level Assembler.

The **SET MAPLIBS** command can be used to setup a list of several datasets, and that list can include a mixture of libraries, sequential datasets, and specific PDS(E) members. Examples:

SET MAPLIBS userid.MYPRODUCT.ADATA
SET MAPLIBS userid.MYPRODUCT.ADATA(TWOSECTS)
SET MAPLIBS hisid.PRODDOC.SEQUENTL

Together, these three commands set up a single MAPLIBS list consisting of the three named datasets and library. For information about the SET MAPLIBS command, see **HELP COMMANDS SET MAPLIBS.**

The xxxMAPLB ddname

MAPLIB lists can be created both by the SET MAPLIBS command and by the presence of an **xxxMAPLB** ddname. Here are the rules regarding Z/XDC's use of the **xxxMAPLB** ddname.

- The **xxxMAPLB** allocation may consist of a concatenation of sequential files or of partitioned libraries, but not of a mixture of both.

- When an **xxxMAPLB** allocation is present, it will be search **last,** after all files and libraries in the currently active list established by SET MAPLIBS commands.

- When **xxxMAPLB** is a concatenation of libraries, it will be processed by Z/XDC as if it were one giant library. I.e., only those members will be searched whose member names either match or relate to the map being searched for. (See below, **How MAPLIBs Are Searched,** for details.)

- When **xxxMAPLB** is a concatenation of sequential files (and/or **specific** library members), it will be processed by Z/XDC as if it were one giant sequential file. I.e., it will be searched through by each and every MAP/DMAP command. No attempt will be made to avoid a search based on the name of the map being searched for.

- There is no Z/XDC command that can allocate or deallocate the **xxxMAPLB** ddname, but when running within TSO, Z/XDC's **TSO** command can be used to issue ALLOCATE and FREE commands to do this.

In the **xxxMAPLB** ddname, **xxx** represents Z/XDC's current clone name. The most common (and default) clone name is **XDC,** but it can be any other legal 3-character name (**V31** for example). For information about Z/XDC clones, see **HELP XDCCLONES.**

How MAPLIBs Are Searched

When the **MAP** command or **DMAP** command starts searching for ADATA, it will know only two things:

- The name of the csect or dsect map that it needs to build, and

- The name of the load module or program object that contains that map.

It will **not** know directly the name of the ADATA library, member, or dataset that contains the ADATA information necessary to build the map. Essentially, Z/XDC will

have to make guesses.

It makes these guesses assisted by lists of ADATA side files and libraries that you establish by use of **SET MAPLIBS** commands.

A **MAPLIBS list** consists of:
- An ordered list of individual datasets and/or libraries set up by one or more **SET MAPLIBS** commands.
- Optionally followed by a concatenation of datasets allocated to the **xxxMAPLB** ddname.

A MAPLIB dataset can be any of the following:
- A sequential dataset
- An Explicitly specified member of a partitioned dataset
- An entire partitioned dataset (i.e. a "library")

Each such dataset or library member may contain either ADATA or GOFF data containing ADATA.

The xxxMAPLB allocation may be a concatenation of sequential files (including files that are explicitly specified members of partitioned datasets). Or it may be a concatenation of entire partitioned libraries. But it may not be a mixture of both.

Z/XDC will search the MAPLIB lists starting with the first dataset in the list and continuing either until the correct ADATA is found or until both the list of datasets and the xxxMAPLB ddname allocation have been exhausted.

For each dataset/library in a MAPLIBS list, Z/XDC will process as follows:

- If the dataset is a sequential dataset, then it will always be searched by all MAP and DMAP commands regardless of the name of the map being searched for. This is because there is no reasonable way to associate sequential dataset names with the names of maps.

- Similarly, if the dataset is an explicitly identified member of a partitioned dataset (example: **dsname(member)**), then it too will always be searched by all MAP and DMAP commands **regardless** of the name of the map being searched for. In other words, the member’s name will not be a factor in deciding whether or not the member should be searched: It will **always** be searched.

- On the other hand, if the dataset is the name of an entire partitioned dataset (i.e. no member name given), then it will be treated as a **library,** and it will be either searched or skipped as follows:

  - If the library contains a member whose name matches the name of the csect map or dsect map being searched for, then that member will be searched.

  - If that search fails, then if the library contains a member whose name matches the name of the load module or program object in which the map is expected to reside, then that member will be searched.

  - No other members will be searched.

Organizing Your ADATA

When a large program containing many csects is built from multiple assemblies, some

programmers will set up the JCL such that the ADATA output for each assembly will be written to separate members of a partitioned dataset, while other programmers will prefer to append all the ADATA together into a single sequential dataset (or PDS(E) member). Z/XDC can accept and process ADATA from either structure: neatly separated by csect into PDS(E) members or piled altogether into a single file.

When ADATA is separated by csect into individual PDS(E) members:

- The **SET MAPLIBS** command needs to indicate to Z/XDC that the entire PDS(E) is an ADATA source (i.e. no member name given... Example: **SET MAPLIBS userid.MYPROD.ADATALIB).**

- The name of each member needs to match the name of the csect that it contains.

- If a member contains ADATA for multiple csects and dsects, only the csect or dsect whose name matches the member name will be findable by the MAP and DMAP commands.

- On the other hand, if the member contains a bunch of dsects, then any dsect can be read with a DMAP command such as the following: **DMAP membername.dsectname.** See **HELP COMMANDS DMAP** for more information.)

When ADATA is piled together into a single sequential dataset or PDS(E) member, that file can be set up as a MAPLIB in any of the following ways:

- If it’s a single sequential dataset, then the dataset can be specified as a MAPLIB, and it will be searched for **all** MAP and DMAP commands. Example: **SET MAPLIBS userid.ADATA.ALLOFIT**

- If the ADATA is all collected into a single member of a PDS(E), then the file can be specified as a MAPLIB in either of two ways:

  - The specific member can be explicitly defined as a MAPLIB. Example: **userid.MYPROD.ADATA(ALLOFIT)** In this case, Z/XDC processes the MAPLIB entry as if it were a sequential dataset. It will be searched for **all** MAP and DMAP commands. In other words, the member’s name will not be a factor in deciding whether or not the member should be searched: It will **always** be searched.

  - Alternatively, the PDS(E) library can be defined as a MAPLIB without explicitly naming the member containing the collected ADATA. Example: **userid.MYPROD.ADATALIB.** In this case, only those members will be searched whose names either match the name of the csect/dsect being mapped or match the name of the load module or program object containing the csect to be mapped.

Recommendations

Use a library for your ADATA, not a sequential dataset.

Write the ADATA for csects into individual members whose names match the csect names.

Write the ADATA for all dsects into a single collective member whose name is something such as "prodDOC" (Example: XDCDOC). Note, the member name can be arbitrary.

Multiple Named MAPLIBs Lists

The SET MAPLIBS command can be used...
- To create any number of **named lists** of MAPLIB datasets,
- Designate which of those lists is currently active,
- Designate which of those lists is the one that will be active by default.
- And permanently save all of your MAPLIB lists into your personal Session Profile.

Subtopics Index

The following topic provides additional information. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

NAMEDLISTS - Creating multiple named MAPLIB lists.

For more information about how Z/XDC searches MAPLIBs, load modules, and program objects, see **HELP MAPS ADATA SEARCHORDER.**

Help MAPs AData MAPlibs Namedlists

The SET MAPLIBS command can be used to build one or more lists of datasets and libraries that the MAP and DMAP commands will search while building source image csect and dsect maps. At any point in time, only one MAPLIBS list can be active. All additional lists that might exist are inactive and so will not be searched by the MAP and DMAP commands.

Having multiple MAPLIB lists makes it a little more convenient for the user to engage in multiple program development projects at the same time. Different projects might require different (and conflicting) MAPLIB libraries. Also, MAPLIB lists are real helpful when managing the simultaneous development or troubleshooting of multiple versions of the same product.

Named MAPLIB lists can be permanently saved in your session’s profile, so once a list has been created, it can be easily accessed whenever required.

Named lists are created by saving an active MAPLIBS list into the session profile and assigning it a name. This is done with the **SET MAPLIBS SAVE name** command. See HELP COMMANDS SET MAPLIBS for more information.

!!!WARNING!!! The SET MAPLIBS SAVE command only saves the MAPLIBS list information into the session’s profile data structure, which is located in private area storage. A **PROFILE SAVE** command must then be used to write the profile data structure out to disk. (This is known as "hardening the data".) If this is not done, then the named list will be forgotten at the end of the debugging session. See HELP COMMANDS PROFILE SAVE for more information.

To activate a previously created named MAPLIBS list, Use the **SET MAPLIBS READ name** command. See HELP COMMANDS SET MAPLIBS for more information.

To display all available MAPLIBS lists, use the **LIST MAPLIBS ALL** command.

To make a default MAPLIBS list that will be automatically made active at the start of a debugging session, create a MAPLIBS list whose name is **DEFAULT.** Example:

```
SET MAPLIBS hlq.XDCV31.XDCADATA
SET MAPLIBS SAVE DEFAULT
PROFILE SAVE
```

Help MAPs AData Searchorder

The MAP and DMAP commands will search for ADATA in the MAPLIB datasets and libraries ahead of looking either in //xxxMAPLB or in the program object or load module. When building a csect or dsect map, the MAP/DMAP commands will search for the ADATA in the following order:

- The datasets and libraries that constitute the currently active MAPLIBS list will be searched from first to last.
- If a //xxxMAPLB allocation exists, then those (one or more) datasets or libraries will be searched.
- If appropriate ADATA is not found in any MAPLIB dataset or //xxxMAPLB allocation, then the program object being mapped is searched.
- If appropriate ADATA is not found in the program object, then the program object or load module is searched for SYM data.
- If appropriate SYM data is not found in the program object or load module, then the MAP or DMAP command fails.

If you wish to shortcut this search order, you can use the **ADATA=** operand on the MAP or DMAP command to direct Z/XDC to look only in the indicated place. This operand also causes the indicated file/library to be added to the currently active MAPLIBS list (so you don't have to specify it more than once).

When a MAPLIB dataset being searched is a sequential dataset, then that dataset will **always** be searched regardless of the name of the map being searched for.

When a MAPLIB dataset being searched is a specific member of a partitioned dataset, then that member also will **always** be searched, again regardless both of the member name and of the name of the map being searched for. This search will occur:

- Regardless of the member's name.
- And regardless of the load module name, program object name, or member name specified on the MAP or DMAP command.

In other words, when a dataset in a MAPLIB list is a specific member of a partitioned dataset, it will be treated **exactly** like a sequential dataset.

When the MAPLIB dataset being searched is a partitioned dataset **without** any particular member specified, then at most two members will be searched:

- **For the MAP command:**

  - If the library contains a member whose name matches an upcased copy of the name of the csect to be mapped, then that member will be searched.
  - If the map is not found there, then Z/XDC will look for a member whose name matches the load module or program object that contains the csect.
  - If either the csect name or load module name or program object name has a syntax that is not legal as a PDS or PDSE member name, then that search is bypassed.

- **For the DMAP command:**
  - If the library contains a member whose name matches an upcased copy of the name of the dsect to be mapped, then that member will be searched first.
  - If the map is not found there, then Z/XDC will look for a member whose name matches the member name given on the DMAP command. If one is found, then that member will be searched.
  - If the dsect name has a syntax that is not legal as a PDS or PDSE member name, then that search is bypassed.

Whenever Z/XDC searches a MAPLIB dataset or member, it caches a copy of that file into storage for use by future searches. For more information, see HELP ∗NEXT (F11) (HELP MAPS ADATA CACHING).

Help MAPs AData CAching

Whenever a sequential dataset or PDS(E) member is searched, a filtered copy of the ADATA contained within that file is saved in storage as a cache. If a subsequent MAP or DMAP command needs to search the same file again, then the cache will be searched instead of the dataset (or member). Caches are manipulated by the following commands:

- **LIST CACHE:** Displays information about the sizes and contents of one, some, or all caches.

- **DELETE CACHE:** Purges from storage one, some, or all caches.

- **DELETE MAPLIBS:** Purges from storage those caches that are associated with the MAPLIBS that are to be removed from the active MAPLIBS lists.

- **LIST MAPLIBS:** Shows which caches (if any) are associated with which MAPLIB datasets.

ADATA is **not cached** when read from program objects.

Note, caches are large. They can take up many megabytes of 31-bit virtual storage. Consequently, whenever Z/XDC itself encounters a shortage of storage, it attempts to make more storage available by automatically purging caches on an LRU (Least Recently Used) basis.

Unfortunately, Z/XDC cannot monitor storage requests done by the user program. Accordingly, the presence of cached ADATA can conceivably cause storage shortages for your program. If this should occur, then use the LIST CACHE command to determine what caches exist and how large they are. Then use the DELETE CACHE command to purge those caches that you do not need.

For information regarding ADATA performance issues, see HELP ∗NEXT (F11) (i.e. HELP MAPS ADATA PERFORMANCE).

Help MAPs AData Performance

ADATA is a comprehensive source of information about assembled programs; however, it also is a **hugely large** source of information! Consequently, generating, saving, and using ADATA presents some problems.

Let me begin with my personal experiences regarding the ADATA and SYM data generated by Z/XDC's own source code. In 2013 I ran some tests where I assembled all 204 source modules and linkedited all 30 load modules that comprise release z2.1 of the Z/XDC product. Here is what I found out.

No SYM data and no ADATA
When I linkedit all of Z/XDC's load modules without either SYM data or ADATA, the load library takes around 209 3390-tracks (a little over 11 megabytes).

SYM data

When I linkedit all of Z/XDC's load modules with filtered SYM data, the load library takes around 1057 tracks (about 56.4 megabytes). That's about a 5 fold increase. (Note, the SYM data had been filtered by XDCSYMED to remove redundant control block definitions. See HELP MAPS SYM EXCESSSYMDATA for more information.)

ADATA embedded in the program object
I have been unable to linkedit Z/XDC modules with embedded ADATA, because when I tried, I found that the linkedit time increased from under a minute to **MORE THAN A DAY!** How much more? I don't know, I had to cancel the job.

So I had to abandon the idea of including ADATA into program objects. Instead, the only feasible way to maintain ADATA is in separate libraries. (This, by the way, is when I realized that I would not be able to avoid writing MAPLIB support.)

Unfiltered ADATA in a separate library
The ADATA for all of XDC's source code requires **more** than **50 thousand** tracks! How much more? I'm not sure, because that's when my jobs started sE37'ing, and I didn't bother to try again. But the gen was pretty close to done when it failed, so I estimate that about 55,000 tracks would have sufficed. That, by the way is a **65 fold increase** over the 848 track increase attributable to SYM data! As I said, ADATA is **massive!**

Filtered ADATA in a separate library
However, when the Assembler generated ADATA was passed through the **XDCADATA** Filtering Program, the size reduction was significant. The DASD space requirement was reduced from 55 thousand tracks down to a around 12 thousand tracks. (YMMV) It's still a lot, but it no longer is unmanageably huge.

(For information about the XDCADATA Filtering Program, see HELP MAPS ADATA EXCESSADATA XDCADATA.)

Conclusions

1) Except for the very smallest programs, PARM='GOFF(ADATA)' should be avoided at all costs. Reasons:

- Linkedit times soar exponentially when ADATA is present in the object file input.

- Unlike SYM data, the Binder does not provide any way I know of to ignore ADATA present in its inputs. (Omitting PARM=TEST to the Binder suppresses SYM data but not ADATA.)

- When ADATA is present, the Binder forces its output module to be a program object, not a load module.

- When the Binder creates a program object, the output library must be a PDSE. (Program objects cannot be written to PDSs.)

2) The disk space required to hold unfiltered ADATA is **huge!** So it is almost mandatory to use the XDCADATA Filtering Program to reduce the Assembler's ADATA output down to a manageable size.

3) Read HELP MAPS ADATA EXCESSADATA XDCADATA for information about the XDCADATA Filtering Program.

Help MAPs AData Excessadata

What is ADATA?

ADATA is a file of data that contains comprehensive (one might even say excessive) information concerning Assembler programs. Z/XDC can use ADATA for producing source level maps of programs and control blocks located in storage. These maps contain the following information:

- The names, locations, lengths, and attributes of **all** statement labels and field labels defined in the Assembler program. This includes **long** labels. (Long labels are not available from SYM data.)

- The names and locations of all Assembler **EQU** symbols, both relocatable and non-relocatable. (EQUs are not available from SYM data.)

- Exact images of all of a program's source statements **(including** all source statements generated by dsect mapping macros regardless of whether or not displays of those statements have been suppressed in the assembly listing!)

- All macro calls as well as indications of which statements following a macro call are macro expansion statements.

- All comment statements.

- For machine instructions, the original object code generated by those instructions.

- All cross reference tables produced during an assembly.

- Way too much other stuff!

Z/XDC can make use of all this information to produce csect and dsect maps that are exceedingly informative. When ADATA is present, storage displayed by the FORMAT and WHERE commands can show the original source code images and commentary for the

programs and control blocks being displayed. In fact, Z/XDC will even recognize when a program's **machine instructions** have been changed and will display **hilighted warnings** about it!

The Problem with ADATA

However... The volume of ADATA can be **massive!** For example, the ADATA files for all of Z/XDC itself (some 200+ assemblies), if I were to generate it all, would require around **55 thousand** tracks on a 3390! For me, that's out of the question... This is where Z/XDC's **XDCADATA filtering program** can be a big help!

XDCADATA Filtering Program

One of the support programs that comes with Z/XDC is named **XDCADATA.** It is an ADATA filtering program that can be used to reduce the size of ADATA files by 50% to 90% or so. It has two strategies for doing this...

- **1)** Z/XDC does not make use of all ADATA record types, so XDCADATA's **type filter** eliminates those types that are unneeded.

  The record types that Z/XDC needs are:
  - 0000: Job Identification Records (Z/XDC doesn't need these, but one of our customers does, so we no longer filter them out.)
  - 0002: Compilation Unit Start/End information
  - 0020: Csect, dsect and entry point information (Some subtypes are discarded.)
  - 0030: Source statement image (Certain statement types are unneeded and are discarded.)
  - 0034: DC/DS information
  - 0036: Machine instruction object code
  - 0042: Individual symbol information
  - 0044: Symbol reference information (Only a handful are needed, so 99%+ are discarded.)

  All other record types are discarded. For Z/XDC, this has resulted in around a 9% reduction in the ADATA volume. (YMMV)

- **2)** The second filter is the **Exclusion list** filter. This is a filtering method that allows you, the programmer, to exclude items (DSECTs, etc) that are in other ADATA files. Typically, these are common layouts that have a separate assembly.

- **3)** The third filter is the **SYMDEL filter.** This is a filtering method that allows you, the programmer, to identify in your source code those areas of the code whose ADATA is to be discarded. When used properly, this filter can eliminate as much as an additional 50% **to 90% of excess ADATA.** Let me explain...

What is Excess ADATA and How is it Created

Often large programs (such as Z/XDC and such as JES2, just to name a couple) are constructed from a large number of separately assembled control sections (csects). Typically each such assembly includes the assembly of a full set of both System and

private control block maps needed for maintaining a consistent understanding of those control blocks across all assemblies.

These maps can run to several thousands of lines of code. The result is that each ADATA file for each of the assemblies will be around 50% to 90% identical to all of the other ADATA side files! In Z/XDC's case, this redundancy, when repeated across some 200+ separate assemblies, results in an ADATA library that **bloats** out to some **55 thousand** tracks on 3390 DASD!

But when that redundant ADATA is identified and filtered out, well in Z/XDC's case, the ADATA library size can shrink down by as much as 80% to (in Z/XDC's case) around 12,000 tracks.

By the way, please don't think that PRINT OFF or PRINT NOGEN will somehow reduce the ADATA that is produced. Whether or not the control block maps show up in the listing has no affect whatsoever on the ADATA. It all is still generated.

Eliminating Excess ADATA using an exclusion list

XDCADATA removes unwanted ADATA if the containing DSECT, CSECT, or RSECT name matches an entry on a supplied exclusion list.

Note that the use of this list means that you do not need to make any changes to the Assembler source. The relevant ADATA records are removed during XDCADATA processing.

The format and use of an exclusion list is fully explained in **HELP MAPS ADATA EXCESSADATA XDCADATA**

Eliminating Excess ADATA with SYMDEL Filtering

XDCADATA also removes unwanted ADATA by scanning for certain triggering symbols that you have to add to your Assembler source programs at various points. These symbols are:

SYMDEL - Start deleting ADATA. (nestable)
SYMUNDEL - Cancel (unnest) a corresponding prior SYMDEL. If no SYMDELs remain nested, then stop deleting ADATA.
SYMNODEL - Stop deleting ADATA. All nested SYMDELs are canceled.

The best way to use these **ADATA Management Triggers** is to place them in the name field of DSECT statements. For example, when you have a section of code for which you want to have the ADATA deleted, place a **SYMDEL DSECT** statement at the start of that code and a **SYMUNDEL DSECT** statement after it.

Then follow the SYMxxDEL DSECT statement with a suitable CSECT, DSECT, RSECT or LOCTR statement to resume the section that the SYMxxDEL statement interrupted.

If you are writing dsect mapping macros, then you could add ADATA Management Triggers into the macros both before and after the dsects that the macros generate. You could add a SYMDEL DSECT statement just ahead of a dsect being generated, and a SYMUNDEL DSECT statement immediately following the dsect's last statement.

What the SYMDEL/SYMUNDEL/SYMNODEL Triggers Do

The SYMDEL, SYMUNDEL and SYMNODEL triggers are processed by the XDCADATA filtering program as follows:

- Every time a **SYMDEL** symbol is encountered, a nesting counter is incremented, and ADATA either begins to be or continues to be suppressed.

- When a **SYMUNDEL** is encountered, the nesting counter is decremented by one (but not if it already is zero). If the counter reaches zero (or remains at zero), then ADATA suppression is discontinued; otherwise, suppression continues.

- When a **SYMNODEL** is encountered, the nesting counter is forced to zero, and ADATA suppression is unconditionally discontinued.

SYMDEL/SYMUNDEL pairs can be used in macros, for example, when you want to temporarily suppress ADATA and then restore the prior suppression state without having to know what that state was.

SYMNODEL can be used to cancel ADATA suppression unconditionally and regardless of the SYMDEL nesting level.

The @SYMOFF and @SYMON Macros

These macros are a way to emit SYMDEL and SYMUNDEL dsects (respectively) in as easy and nondisruptive a way as possible. (There is no macro for emitting SYMNODELs.)

When @SYMOFF and @SYMON bracket a section code, they emit the SYMDEL and SYMUNDEL triggers that will cause the XDCADATA and XDCSYMED utilities to purge all ADATA and SYM data emitted by the bracketed code. The triggers are emitted silently so as not to clutter up your assembly listings.

The macros accept almost no operands. If any are given, they are mostly ignored. (Ditto for the name field.) For complete usage details as well as several examples, see **HELP MACROS @SYMOFF/ON.**

A Note About Excess SYM Data

The above excess ADATA data problem also exists for SYM data, but like ADATA data, excess SYM data can be controlled by proper use of SYMDEL and friends. See **HELP MAPS SYM EXCESSSYMDATA** for more information.

More Information

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

EXAMPLES - Examples and suggestions regarding the usage of the ADATA Management Triggers.

XDCADATA - Using the XDCADATA filtering program to remove excess ADATA.

Help MAPs AData Excessadata Examples

Typically, excess ADATA occurs when a complex program is created from several, separately assembled csects. The ADATA that is generated will, collectively, contain redundant information from the common mapping macros that occur in each of the separately assembled CSECTs. In the worst (and not uncommon) case, this could be as many as one copy of each dsect's ADATA from each assembly... all gathered together into your now bloated ADATA library.

As described in detail in the prior topic **(HELP MAPS ADATA EXCESSADATA),** excess ADATA can be managed by using the special symbols **SYMDEL, SYMUNDEL** and **SYMNODEL** on DSECT statements appropriately inserted into your source code. Here is an example.

A Typical Case

Typically, when a program or product is built from multiple assemblies, most assemblies will include a dsect maps definition section for both System control blocks as well as for control blocks that are private to the program. In all assemblies but one, insert a **SYMDEL DSECT** statement ahead of your collection of mapping macros, and then a **SYMUNDEL DSECT** statement afterwards, as shown below. (Notice that after the SYMUNDEL DSECT, you probably have to then resume your program's csect.) Example:

```
MYPROGRA CSECT ,
         ...
SYMDEL   DSECT ,                DELETE ADATA
         PRINT   NOGEN          HEY, LOOK IT UP
         IHAASCB ,              ASCB MAP
         CVT     DSECT=YES,PREFIX=YES,LIST=YES
         DCBD    DSORG=(PS,PO)  DCB MAP
         IHAPSA  ,              PSA MAP
         IKJTCB  LIST=YES       TCB MAP
         PRINT   ON,GEN,NODATA  RESUME NORMAL DISPLAY
SYMUNDEL DSECT ,                STOP DELETING ADATA
MYPROGRA CSECT ,                RESUME NORMAL CSECT
         ...
```

After assembling the above, you would then run the Assembler's //SYSADATA output file through the **XDCADATA** filtering program. Then all of the ADATA generated for all of the dsect maps occurring between the SYMDEL and SYMUNDEL statements will be completely eliminated from the ADATA file! This can lead to a **substantial reduction** in the DASD space required for your ADATA library.

Please notice the following:

- Generally a SYMDEL, SYMUNDEL and SYMNODEL DSECT statement should be followed immediately by another CSECT, DSECT, RSECT or LOCTR statement so that correct dummy section and control section construction is not disrupted. You may have to do this yourself (as shown above following the SYMUNDEL DSECT statement).

- As many SYMDEL, SYMUNDEL and SYMNODEL DSECT statements may be used as needed. Thus, scattered groups of ADATA can be deleted easily.

- The SYMDEL, SYMUNDEL and SYMNODEL symbols can be used on any type of Assembler statement, but only DSECT statements permit repeated occurrences of a symbol, so using them on DSECT statements is most convenient.

An Alternative to SYMDEL and SYMUNDEL

@SYMOFF and **@SYMON** are a pair of macros that can be used instead of SYMDEL and SYMUNDEL (respectively) for filtering unwanted ADATA and SYM data from the Assembler's output files. They are a little easier to use than SYMDEL and SYMUNDEL, and they are designed to reduce the visual clutter in assembly listings. For usage details and examples, see **HELP MACROS @SYMOFF/ON.**

A Suggestion

As mentioned above, you should delete the excess ADATA from all assemblies but one. Let us suggest that the one assembly that you do not delete the ADATA from be a special **DOC** assembly (similar to the HASPDOC assembly used by JES2, or similar to the XDCMAPS assembly distributed with the Z/XDC product).

In other words, create a special assembly that has nothing but all of the control block maps for your program. You can then linkedit that assembly into a separate **xxxxxDOC** (for example) load module and save its ADATA into a **xxxxxDOC** member of your ADATA library. Then Z/XDC's **DMAP** command can use the xxxxxDOC member as a source for the dsect maps for all of your program's control blocks. (Example: **DMAP xxxxxDOC.MYCBLOCK**)

Help MAPs AData Excessadata Xdcadata

One of the utilities that we provide with Z/XDC is named **XDCADATA.** It is an ADATA Filtering Program that can be used to reduce the size of ADATA files by **50%** to **90%+** or more.

XDCADATA Filtering Strategies

XDCADATA has three strategies for removing ADATA:

- **1)** Z/XDC does not make use of all ADATA record types, so XDCADATA uses an internal **Type Filter** to eliminate those types that are unneeded.

  The record types that Z/XDC needs are:
  - **0000:** Job Identification Records (Z/XDC doesn't need these, but one of our customers does, so we no longer filter them out.)

- **0002:** Compilation Unit Start/End information
- **0020:** Csect, dsect and entry point information (Some subtypes are discarded.)
- **0030:** Source statement image (Certain statement types are unneeded and are discarded.)
- **0034:** DC/DS information
- **0036:** Machine instruction object code
- **0042:** Individual symbol information
- **0044:** Symbol reference information (Only a handful are needed, so 99%+ are discarded.)

All other record types are discarded.

For Z/XDC, this has resulted in a minor reduction or 9% or so. (YMMV)

- **2)** The second filter is the **Exclusion list.** This is a filtering method that allows you, the programmer, to exclude items (DSECTs and such) that are redundant across multiple assemblies. See **HELP DDNAMES XCLUD** for details about creating an exclusion file.

- **3)** The third filter is the **SYMDEL Filter.** This is a filtering method that allows you, the programmer, to identify in your source code those areas of the code whose ADATA is to be discarded. When used properly, this filter can eliminate as much as an additional 50% to 90% of excess ADATA.

The Filtering Triggers are DSECT statements having the names **SYMDEL, SYMUNDEL** and **SYMNODEL.** Usage for these triggers is explained in detail in a topic named **HELP MAPS ADATA EXCESSADATA.**

Compression

In addition to filtering, XDCADATA automatically compresses the output ADATA using an **LZW** algorithm.

Compression is on by default. It can be turned off by adding **//XDCNCOMP DD DUMMY** to the JCL. (You will need to turn it off if you have other ADATA consumers besides Z/XDC.)

Making a DOC Assembly

For your private control blocks, you may want to create a **DOC** assembly as the sole assembly where their ADATA is not filtered out. This, then, will become a repository for maps for your private control blocks.

Such a DOC assembly need not include z/OS control block maps since we provide ADATA for those in our own repository named **XDCMAPS.** See **HELP MAPS XDCMAPS** for more information.

How to Run XDCADATA

XDCADATA is designed to be completely flexible in the manner in which you use it.

Generally, if it makes sense, you can do it. So XDCADATA can be invoked in any of the following ways:

- **Within the Assembler:** XDCADATA can be invoked by the High Level Assembler as an ADATA Exit. To do this, you will need to provide the following PARM Options to the Assembler:

  - **EXIT(ADEXIT(XDCADATA))**
    This causes the High Level Assembler to call XDCADATA as a processing exit for each and every ADATA record that it generates. XDCADATA will then either preserve or discard each record. See below for details.

  - **ADATA** and/or **GOFF(ADATA)**
    These cause the Assembler to generate ADATA. without them, there would be nothing for XDCADATA to do.

  Performance Note! The call overhead between the Assembler and ADATAEXITs, while necessary, **is considerable.** Accordingly, when incorporating XDCADATA into the build process for large numbers of assemblies, you probably will want to invoke XDCADATA in a **separate jobstep** following the assembly steps (rather than invoking XDCADATA as an assembly time exit). See the following.

- **In JCL:** XDCADATA can be invoked as a normal program in JCL in a jobstep following an assembly step. In this case, it will read ADATA from an assembler created **//SYSADATA** file, and write its filtered output to a **//XDCADATA** file for permanent storage. The input file (//SYSADATA) can be either an ADATA file or a GOFF file containing embedded ADATA.

  Performance Issue! From a performance point of view, running XDCADATA as a separate jobstep is **far better** then running it as an HLASM exit! This is because of necessary but rather considerable overhead in the Assembler's Call Interface to ADATAEXIT routines.

- **In TSO:** XDCADATA can be invoked in TSO either as a normal program or as a TSO Command. In this case (as in the JCL case), XDCADATA will read ADATA from a **SYSADATA** file, and write its filtered output to an **XDCADATA** file. The input file (SYSADATA) can be either an ADATA file or a GOFF file containing embedded ADATA.

Control Parameters

This is easy. There are none. XDCADATA takes all of its cues from examining its environment.

DDNAMEs

There are five DDNAMEs that XDCADATA cares about:

- **//SYSADATA** This is an input file containing raw ADATA from the Assembler.

- **//XDCADATA** This is an output file to which XDCADATA writes filtered and compressed ADATA.

- **//XDCPRINT DD SYSOUT=*** XDCADATA writes its information and processing messages to this file.

- **//XDCXCLUD** This is an optional file that points to a list of dsects to be removed from the ADATA. For details, see **HELP DDNAMES XCLUD.**

- **//XDCNCOMP DD DUMMY** This is a keyword ddname that, when present, turns off compression. You probably want to do this if you have ADATA consumers other than Z/XDC.

//SYSADATA

Whether or not XDCADATA reads this file depends on whether XDCADATA is running as a utility or as an Assembler Exit...

- When running as an Assembler ADATA Exit, XDCADATA receives all of its input ADATA records directly from the Assembler; therefore, it ignores the //SYSADATA file (Good thing too!)

- On the other hand, when XDCADATA is running as a utility (or TSO command), XDCADATA requires the //SYSADATA file as the input source for the ADATA to be filtered. In this case, the //SYSADATA file must be present and it must have the following characteristics:

  - The data within the file may be either ADATA of GOFF records.

  - When //SYSADATA contains GOFF records, XDCADATA will extract all embedded ADATA that it finds and ignore the rest.

  - When //SYSADATA contains ADATA...

    - Each file record may contain a discrete ADATA record.

      This of course is suitable for ADATA as produced by the High Level Assembler.

    - Or the ADATA may be streamed into the file without regard to file record boundaries. I.E. If an ADATA record ends before the end of a file record, then the next following byte must be the start of the next following ADATA record, and ADATA records may span across any number of file records.

      This is suitable for ADATA that has been generated by a "Cross Assembler" and then uploaded to the mainframe into an arbitrary sequential file. Note, Tachyon Software's **z/Assembler** is one such Assembler. See **HELP TACHYON** for more information.

  - The //SYSADATA file may have any RECFM and any reasonable LRECL. XDCADATA's support for streaming ADATA records across multiple file records is not affected by the RECFM. The streaming support will work regardless of whether RECFM =F, =V or =U, and regardless of whether or not QSAM's spanned records support (RECFM=VBS) is or is not selected.

- The //SYSADATA file may be a concatenation of any number of sequential datasets, and...
  - The concatenation may be a mixture of ADATA files and GOFF files,
  - The concatenated files can all have differing RECFMs and LRECLs.

  It's all good.

//XDCADATA
When this file is present, the XDCADATA program writes the filtered ADATA out to it. Whether this file is **optional** or **required** depends upon whether **or not** XDCADATA is running as an Assembler ADATA Exit...

- When XDCADATA is running as an ADATA Exit, the //XDCADATA output file is optional:
  - When //XDCADATA is absent, the XDCADATA program filters the ADATA that is returned to the Assembler. The Assembler will then write the filtered ADATA out either to it's //SYSADATA file or to GOFF records (or to both).
  - On the other hand, when the //XDCADATA file is present...
    - The XDCADATA program writes the filtered ADATA out to the //XDCADATA file,
    - But it also returns **all** ADATA **unfiltered** back to the Assembler! So the Assembler's //SYSADATA file (and/or GOFF file) receives all ADATA just as if the XDCADATA program did not exist.
- When XDCADATA is running as a utility (or TSO command), the //XDCADATA output file is required, and the filtered ADATA is written to it.

  The //XDCADATA file must have the following characteristics:
  - Its **RECFM** must be **VB.**
  - XDCADATA does not span or stream its output file, so //XDCADATA's **LRECL** must be large enough to accommodate the longest surviving ADATA record. **LRECL=27994** is recommended. (Note, XDCADATA's Processing Report will report the longest ADATA record written. The longest I've seen has been 8,183 bytes.)
  - the //XDCADATA file will usually be compressed. If compressed, the header part of each record is not compressed.

    Compression may be prevented by the presence of the **//XDCNCOMP** file (as documented below).

//XDCXCLUD
If this file is present, it is read by the XDCADATA program to build an exclusion list of dsect names to be removed from the ADATA.

For details about constructing an exclusion list, see **HELP DDNAMES XCLUD.**

//XDCNCOMP DD DUMMY
The presence or absence of this file controls whether or not the ADATA written to the //XDCADATA file is compressed.

The **default action is to compress** the ADATA, but when this ddname is present, compression does not occur.

Normally, you will want compression to occur, but if you have ADATA consumers other than Z/XDC, then you will want to suppress compression.

//XDCPRINT DD SYSOUT=*
This file is optional.

- When present, the XDCADATA program writes a concise but still fairly comprehensive processing report to it.
- When absent, XDCADATA writes its report to an alternate message stream as follows:
  - When running **in TSO,** the report is written to the user's terminal (via PUTLINE).
  - When running **in TSOBATCH,** the report is written to //SYSTSPRT (via PUTLINE).
  - When running **in normal batch,** the report is written to the JES job log via WTP (WTO ...,ROUTCDE=11).

When the //XDCPRINT file is present, it should have the following characteristics:

- It may have any RECFM. The default will be RECFM=VB.
- It may have any reasonable LRECL. The default will be:
  - LRECL=132 for RECFM=F or =U.
  - LRECL=136 for RECFM=V

  When the LRECL is too small, long messages will be wrapped.

Additional DDNAMEs

//XDCADUMP
When this file is present, XDCADATA writes to it text messages showing each and every ADATA record that it read or received, and what it decided to do with each record. Exactly one message will be written for each record. (Long messages will be truncated.) Needless to say, this report can be huge.

When the //XDCADUMP file is present, it should have the following characteristics:

- It may have any RECFM. The default will be RECFM=VBA.
- It may have any reasonable LRECL. The default will be:
  - LRECL=133 for RECFM=F or =U.
  - LRECL=137 for RECFM=V

  When the LRECL is too small, long messages will be truncated.

DDNAMEs and Z/XDC Clone Names

A Z/XDC **Clone** is an instance of Z/XDC whose name has been changed from **XDC** to some other three-character name. For full information about Z/XDC Clones, see **HELP XDCCLONES.**

Just as happens throughout the rest of Z/XDC, when you run an instance of XDCADATA that has been renamed to some clone name (Example: **V31ADATA),** then those DDNAMEs that xxxADATA looks for that start with "XDC" also change to match the clone name. Specifically...

- //XDCPRINT becomes //xxxPRINT.
- //XDCADUMP becomes //xxxADUMP.
- //XDCADATA becomes //xxxADATA.
- //XDCNCOMP becomes //xxxNCOMP.
- //XDCXCLUD becomes //xxxXCLUD.

But...

- //SYSADATA remains the same.
- //XDCISxxx remains the same. (Notice that there is no requirement for [or prohibition against] the debugging clone being the same as the xxxADATA clone.)

Suggestion: Do not try to use **SYS** as a clone name.

Performance Considerations

Important! When XDCADATA is invoked as an ADATAEXIT by the High Level Assembler, it has to be called each time for each and every ADATA record that the Assembler generates. For large product builds (like Z/XDC) involving hundreds of assemblies, this can add up to millions of calls!

The call interface between the Assembler and an ADATAEXIT has considerable overhead on **both** sides of the interface. In XDCADATA’s case, this overhead includes the following:

- XDCADATA has to get and free a temporary work area on each call.
- XDCADATA has to obtain and then cancel an ESTAEX on each call.
- On each call, XDCADATA has to go through some dozen or two lines of code first to detect that it is being called by HLASM and then to locate its persistent data areas anchored in the HLASM plists.
- HLASM itself cancels its ESTAEX for each exit call and then reinstates it for each return.
- I do not know what other per call processing HLASM is doing, but apparently it is a significant amount.

All this has the net effect of roughly **doubling** both the CPU time and the wall clock time taken by each individual assembly!

On the other hand, when XDCADATA is run as a separate jobstep, it is only called once to process the entire ADATA file, so all of the per-call setup/takedown is avoided; and in fact, XDCADATA’s processing time drops to something negligible!

Help MAPs AData MACros

When a FORMAT, FIND, WHERE or EWHERE command is used to display storage that is covered by a source image map (such as an ADATA map), the display that is produced will look very similar to what you would see if you were looking at the program's source listing. You would see the program's statements as they were originally written, you would see all of your commentary, and you would see plus signs (+) on statements that were produced by macro expansions and equal signs (=) on statements that were produced by COPY code.

With respect to code generating macros, Z/XDC allows you to control whether their expansions are included in the display or are hidden from view. This is done by use of operands on the **SET FORMAT** command, the **FORMAT** command, the **WHERE** command, the **EWHERE** command and the **FIND** command. The operands are:

- **SHOWMCODE:** Macro expansions are displayed.
- **HIDEMCODE:** The display of macro expansions is suppressed.
- **CURRENTMCODE:** The display of macro expansions is suppressed **except** if the macro expansion currently being displayed includes the error level or retry level PSW address. In this case, the expansion is shown.

The SHOW/HIDE/CURRENTMCODE operands do **not** affect displays produced by dsect maps that map data areas and control blocks. Their expansions are always displayed regardless of the SHOW/HIDE/CURRENTMCODE setting.

The point of all this relates to structured programming macros. The CURRENTMCODE setting is intended to make code developed using structured programming macros easier to read.

For more information about the commands, see:

HELP COMMANDS SET FORMAT
HELP COMMANDS FORMAT
HELP COMMANDS WHERE
HELP COMMANDS EWHERE
HELP COMMANDS FIND

When:

- **HIDEMCODE** is in effect,
- And a FORMAT, WHERE, EWHERE or FIND command is issued to display storage to which a source image map applies,
- And the display includes machine instruction source code from a macro expansion,

Then all statements from the macro and most storage falling within the range of the macro will not be displayed. About all that you would see would be the following:

- If the PSW points into the macro expansion, then the instruction pointed to by the PSW will be displayed.

- If an equate name, a csect name, or any other name **(not** arising from the overlaying map) occurs within the macro expansion, then the object code labeled by that name will be displayed.

- If a register points into a macro expansion, then the data or machine instruction being pointed to will be displayed.

When **SHOWMCODE** is in effect, then nothing is suppressed.

When **CURRENTMCODE** is in effect:
- If either the retry level or error level PSW points into a macro expansion, then that expansion is displayed as if SHOWMCODE were in effect.
- Otherwise (i.e., if no PSW points into the macro expansion), then that expansion is suppressed as if HIDEMCODE were in effect.

The **SHOWMCODE/HIDEMCODE/CURRENTMCODE** operands affect only storage displays mapped via **ADATA,** not those mapped via SYM data. Also, they do **not** affect the display of source images generated from data area or control block mapping macros. Those images are always displayed regardless of the **cccccMCODE** setting.

The **SHOWMCODE/HIDEMCODE/CURRENTMCODE** operands affect only what is displayed. In particular, they do **not** affect the **actions** of the various TRACE commands. **TRACE** and **TRACE BY** remain machine instruction driven. (They do not become macro driven.)

Help MAPs AData Tips

Excess trash at the Start of Csect Maps

One common technique, when writing Assembler programs that use lots of control blocks, is to code the mapping macros for those cblocks at the start of the assembly ahead of the program's actual code. But it turns out that for many IBM written mapping macros, a whole lot of commentary is included at the start of the macro, **ahead** of the dsect card. This means that, as far as the Assembler is concerned, all of that commentary belongs, not to the dsect being described, but to the control section (usually) from which the mapping macro is being invoked. That means when Z/XDC's MAP command is issued to load the ADATA map of that control section, a whole lot of trash from these mapping macros will be included in the control section map, assigned to the start of the program. That's real ugly, and that's real inconvenient!

Here's what you can do to eliminate that problem. Add an **IGNORE DSECT** statement to your program ahead of all of the mapping macros. Then code your normal CSECT statement following the mapping macros. That way, all the excess trash from the mapping macros is assigned by the Assembler to the IGNORE dsect and not to your control section. This greatly improves the map at the start of your program. (Note: "IGNORE" is just an appropriate name. There's nothing special about it, and you are free to choose any unused name you prefer.)

Help MAPs DWarf

Z/XDC's **c/XDC** Licensed Feature provides support for debugging programs written in XL C/C++ and Metal C and which are compiled by the XL C compiler. (Z/XDC does not offer support for C programs compiled using other compilers.) See HELP SUPPORT FEATURESANDCAPS for more information about Licensed Features.

Z/XDC builds XL C/C++ and Metal C program maps from various mapping and debugging information that is created by the XL C compiler when **PARM=DEBUG** is specified for

the compilation.

When **PARM=DEBUG** is specified, the XL C compiler does the following:

- The compiler suppresses object code optimization. This is so that the generated object code will have a clear relationship to XL C/C++ and Metal C program source statements.

- The compiler writes a **DWARF** data file to ddname //SYSCDBG (which must point to a sequential FB-80 dataset or PDS(E) library member). This file contains debugging information about the program including the names of all of the program source files that were read by the program compilation process.

- The compiler causes the resulting XL C/C++ and Metal C program object to contain the name of the DWARF file dataset.

!!!WARNING!!! Be aware that the debugging information produced by the XL C compiler includes the dsnames (and path names as appropriate) both of the DWARF file and all of the XL C/C++ and Metal C program's source files! This information is **hard-coded** into both the program object and the DWARF data. This means:

- Either care must be taken to ensure that both the DWARF data file as well as all source program files not be moved, renamed or changed,
- Or if you do move or rename the DWARF and/or source files, then you will need to use Z/XDC's **LIBRARYLISTS** command to let Z/XDC know where it can find the necessary files.

For more information, see HELP DEBUGGING C.

Help MAPs Floating

After a dsect map has been read from a library, but before it can be used to reference storage, a **USING** command must be issued to tell Z/XDC what storage that map is to represent. (See **HELP COMMANDS USING** for details.) Once that is done, the symbols within the map can be used in address expressions. Also, when the mapped area of storage is displayed via the FORMAT or WHERE command, the map's symbols are automatically used to label the fields and instructions being displayed.

The **USING** command can be used to assigned either a **fixed** or **floating** base to any dsect map:

- When a **fixed base** is assigned, the address expression that you give for the base address is evaluated immediately, and the result is assigned as the map's base. The map will then continue to represent that particular location, regardless of whether or not the control block being represented is itself deleted. The map will remain until it is either **DROP'd, DELETE'd** or reassigned via another **USING** command.

- When a **floating base** is assigned, the address expression that you give for the base address is **not** immediately evaluated. Instead, it is saved as part of the map's definition, and it is automatically reevaluated each and every time the map might be needed. Specifically:

    - The basing address expression is evaluated every time any of the map's

symbols are used in address expressions.

- It is also evaluated every time storage is displayed (via the **FORMAT, WHERE,** and other commands) just in case the map might currently represent the locations being displayed.

The defining address expression for a floating map may be **any** valid address expression! Examples:

- It might be based on a **register:** For example, **USING MYQNTRY R9% F** assigns a map named **MYQNTRY** to represent whichever queue entry is currently pointed to by general register R9, **regardless** of how often the pointer in R9 is changed by the user program.

- It might be based on a changing pointer in **storage:** For example, **USING TCB 21C% F** causes the **TCBFIX** map to represent the current task control block, regardless of under which task Z/XDC gains control in a multi-tasking program.

- It might be based on a field located within another **floating map!** For example, **USING RB .TCBRBP% F** causes the **RB** map to always represent the newest RB queued from whichever TCB is currently represented by the map that contains the **TCBRBP** field. This will be true even if the TCB map is moved from one TCB to another and even if the queue of RBs changes from moment to moment.

The address expression upon which a floating map is based may, itself, reference other floating maps and equates. By this means, an arbitrarily complex structure of floating maps and equates can be created, all being ultimately dependent upon just one anchoring pointer. Then if that single pointer should happen to change, the **entire** structure will automagically move in reaction to that single change.

An Example

The following commands illustrates how to create a complex structure of maps and equates and having the entire structure automagically float from one instance to another simply by changing the anchoring location of a single, root map or equate. That anchoring point can be changed either manually (by command) or automatically (when the anchoring data in storage is changed by programming). It just depends upon how you choose the anchor.

This example loads dsect maps for a number of z/OS system control blocks and then links them together via their documented field names and pointer fields.

The root map will be for an **ASCB** (Address Space Control Block). When that map is reassigned from one ASCB to another, the entire floating structure will automatically move from representing one set of control blocks in one address space to those in another.

Let’s start by loading the dsect maps.

```
DMAP XDCMAPS.ASCB
DM          .ASXB
```

DM .TCB [Follow the link to see about nicknames]
DM .JSCB [ditto]
DM .PSCB
DM .UPT
DM .RLGB
DM .ECT
DM .LWA
(Note the extra spaces are not required. I'm just using them to make things look pretty.)

These **DMAP** commands load dsect maps for several system and TSO related control blocks from a load module named **XDCMAPS.** This is a module what we include as a part of the Z/XDC product. It contains nothing but the assemblies of nearly **three thousand** IBM control block mapping macros. For more information, see **HELP MAPS XDCMAPS.**

DMAP JSCB IEZJSCB
DMAP PUPT UPT
DMAP RECT ECT
DMAP LECT ECT
These commands make copies of some of the maps that were just loaded.

USING ASCB @ASCB FLOAT
This command assigns the **ASCB map** to represent the Address Space Control Block labeled by the **@ASCB built-in equate.** @ASCB initially labels the ASCB for the current address space. However, if you use the **SET ASID** command to change Z/XDC's focus to another aspace, then @ASCB will automatically move to represent that new aspace's ASCB. And since the above command **floats** the map over the equate, the map will automatically move also!

USING ASXB ASCB.ASCBASXB? FLOAT
USING LWA ASXB.ASXBLWA? FLOAT
USING LECT LWA.LWAPECT? FLOAT
USING TCB ASCB.ASCBXTCB? FLOAT
USING JSCB TCB.TCBJSCB% FLOAT
USING PSCB JSCB.JSCBPSCB% FLOAT
USING PUPT PSCB.PSCBUPT? FLOAT
USING RLGB PSCB.PSCBRLGB? FLOAT
USING RECT RLGB.RLGBECT? FLOAT
These commands build a structure of floating maps that maps several z/OS and TSO related control blocks. The details of what these control blocks mean are not important here. (You can look them up in IBM manuals if you're curious.) What is important is that a pretty complex and interesting floating structure has been created, all based upon a built-in equate named **@ASCB.**

For example: let's take a look at how the floating location of the **RECT** is determined:
- The **RECT** map is dependent upon the location of the **RLGB.**
- The **RLGB** map is dependent upon the location of the **PSCB.**
- The **PSCB** map is dependent upon the location of the **JSCB.**
- The **JSCB** map is dependent upon the location of the **TCB.**

- The **TCB** map is dependent upon the location of the **ASCB.**
- The **ASCB** map is dependent upon the location of the **@ASCB** equate.

- The **@ASCB** equate represents the location of the Address Space Control Block for the foreign or local address space that Z/XDC is currently targeting. (See **HELP VIRTMEM XDCACCESS** for information about Foreign Address Space Mode.)

If the resolved location of the **@ASCB** equate should change, then the resolved locations of all of the dependent maps will automatically change also!

The resolved location of @ASCB **will** change if you use the **SET ASID** command.

@ASCB will always point to the particular Address Space Control Block from which the target address space is anchored. (see **HELP EQUATES BUILTIN CBLOCKS.)** Thus, whenever the **SET ASID** command is issued, all of the dsect maps that are anchored from @ASCB will automatically move to point to those instances of those control blocks that reside in the new target address space.

BTW: This particular example is useful enough that the Z/XDC product distribution includes a script consisting of the commands shown above. That script is named **TSOMAPS.** For more information about scripts, see **HELP SCRIPTS OURSCRIPTS.**

More Information

For additional information about setting basing addresses for dsects, see **HELP COMMANDS USING.**

For related information pertaining to **floating equates,** see **HELP EQUATES FLOATING.**

Help MAPs AUTOCloning

Autocloning is a method by which you can automatically generate an entire series of equates or dsects maps to represent either each entry in a table or each control block on a chain. The equates or dsects that are created all have unique sequence numbers but share a common root name that you must provide.

Chains

The **EQUATE** and **USING** commands have operands that let you define the following about chains:
- Where the chain field is.
- How wide the chain field is (3, 4, or 8 bytes).
- A mask for selecting a chain field’s significant bits.
- The chain field value that signifies end-of-chain (zero, usually).
- The maximum number of chain entries to label or map.

Tables

Z/XDC’s autocloning supports tables of contiguous entries containing either fixed

length or variable length entries. When variable, the table entries will (presumedly) have a length field. This length field can be of any reasonable width, and it may define either the table entry's entire length or only the length of the entry's variable portion (in which case, all entries in the table are presumed to consist partly of common fields of a fixed size). In support of all this, the **EQUATE** and **USING** commands have operands that let you define the following about tables:

- For fixed length table entries, or for variable table entries containing a fixed length root section, you can specify the length either of the entries or of the root section.
- For variable length table entries, you can specify where the length field is and how wide it is (1 through 8 bytes).
- You can specify either the length field value (if any) that identifies the end of the table or the address at which the table ends.
- Finally, you can specify the maximum number of table entries to label or map.

Whenever a series of equates or maps is created via autocloning, Z/XDC first checks to see whether or not a series of previously created equates or maps, having the same root name, already exists. If so, then that series is purged in its entirety prior to the creation of the current series.

The equates or dsects that are generated through autocloning can all have the same attributes and characteristics that any other equate/dsect might have. In particular, they can be assigned either fixed bases or **floating bases,** thus the equates/dsects can be created to represent either particular instances or generic instances of table or chain entries.

Floating maps and equates are a very powerful and useful capability, but you need to be aware that the resolution of floating bases is extremely **CPU intensive** and has to be done repeatedly during typical Z/XDC processing. Therefore, if the number of equates and maps having floating bases grows "too large", then **Z/XDC's performance can be impacted severely**!

Unfortunately, the autocloning process has the potential of creating very large numbers of equates and maps. If it is used to create a **large number** of floating maps or equates, then Z/XDC's responsiveness may become unacceptably sluggish. So the **FLOAT** operand should be used with great care! Typically, you probably should never create more than a few dozen floating equates and maps, in total.

Help MAPs Xdcmaps

One of the modules distributed with the Z/XDC product has no code in it, but instead provides both the SYM data and ADATA for more than 2800 IBM control block dsects. This module is named XDCMAPS, and its data is available in the following places.

- **SYM Data:** The SYM data versions of the maps are located within the **XDCMAPS** load module itself. The module is available in the load library whose factory default name is **hlq.XDCV31.XDCLINK.**

  - If that library is present in your address space's standard load module search order (JOBLIB, STEPLIB, linklist, etc.), then its maps will be automatically available to the **DMAP** command. (See **HELP COMMANDS DMAP**

LOAD for more information about both the DMAP command and the standard search order.)

- Otherwise, you can still access the SYM data maps using the DMAP command's **SYMDATALIB=** operand.

- **ADATA:** The ADATA versions of the maps are located in the library named **hlq.XDCV31.XDCADATA.** The ADATA has been segmented alphabetically into 26 members named XDCMAPSA, XDCMAPSB, ..., XDCMAPSZ. Each member contains the ADATA for those dsects whose true names start with the same letter that the member's name ends with. (Note that the IHADCB map, for example, will be located in member XDCMAPSI, not XDCMAPSD.) However, all 26 members are processed by the **DMAP** command collectively as if they were a single giant member. (This segmenting was done for performance reasons.)

  - If hlq.XDCV31.XDCADATA is in your **MAPLIBS** list, then this form of the maps will be automatically available to the **DMAP** command.

  - Otherwise, you can still access the ADATA maps using the DMAP command's **ADATALIB=** operand.

(Note, **hlq.XDCV31.XDCxxxxx** are the **factory default** names for Z/XDC's product installation libraries. Ask your Systems Programmer for the actual library names at your Data Center.)

You can use XDCMAPS as a quick and easy source for dsect maps of most common IBM control blocks. Just issue **DMAP XDCMAPS.mapname.**

In general the names of the IBM control block dsect maps are documented in the various "Data Areas" manuals from IBM. A Google search for **z/os data areas** should find them.

Examples:

DMAP XDCMAPS.TIOT
This command will load either the SYM data version of the map or the ADATA version, depending upon whether or not the hlq.XDCV31.XDCADATA library is in the currently active **MAPLIBS** list. (See **HELP MAPS ADATA MAPLIBS** for more information about MAPLIBS lists.)

See **HELP MAPS SYM** for more information about SYM data maps.

DMAP XDCMAPS.TIOT SYMDATA
This coerces the ADATA/SYM data choice towards SYM data.

DMAP XDCMAPS.TIOT ADATALIB=hlq.XDCV31.XDCADATA
This coerces the ADATA/SYM data choice towards ADATA, and it also explicitly tells Z/XDC the name of the ADATA library, thus avoiding having to set up a MAPLIBS list.

Note that under the covers, Z/XDC will automatically add this library to the active **MAPLIBS** list so that you do not have to provide it a second time. See **HELP**

MAPS ADATA MAPLIBS for more information.

USING TIOT 21C%+C%
This command assigns the just loaded TIOT map to the Task I/O Table that exists within the Home Address Space.

FORMAT TIOT
Finally, this command displays the Task I/O Table, formatted according to the TIOT map.

Nicknames, Prefix Sections and Zero Points

For most control blocks in **XDCMAPS,** the **true names** of the maps match their nicknames (e.g. SCB, TIOT, ASCB, PSA, etc.). However, for many of the historically older control blocks, the true names are less obvious. When you don't know a cblock map's true name, you may have to resort to looking it up in the various **Data Areas** manuals from IBM.

You can find the **Data Areas** manuals online using a simple Google search. **z/os data areas** should do the trick.

Control Block Prefix Sections

Some z/OS control blocks (especially some of the older ones) have **prefix sections.** These are fields in the control block that **precede** the map's normal **zero point** (the location in the control block to which other cblocks point). Accessing the prefix fields from the zero point requires **negative offsets.**

When building the **XDCMAPS** load module, whenever a cblock had a prefix section, we chose to assemble it. Unfortunately, this causes the map's field offsets to be "wrong", because as far as the Assembler is concerned, there is no such thing as a "negative" offset, so the first field in the prefix would be assigned offset 00000000 (instead of minus-whatever). Here's an example:

- For the CVT, the field that is pointed to by storage location **X'00000010'** is named **CVTTCBP.**
- Consequently, the CVTTCBP field is considered to be the cblock's **zero point.**
- However, CVTTCBP is **not** the map's first field. There is a **prefix section** named **CVTFIX.**
- The prefix section for the CVT is X'100' bytes long.
- Consequently, absent a **zero point adjustment,** the CVTTCBP field would be displayed as being at offset X'00000100', which is confusing.
- The **DMAP** command has a **ZEROPOINT=** operand that can be used to adjust a map's zero point at the time it is built. See **HELP COMMANDS DMAP LOAD** for the details.

Z/XDC's Support for Nicknames and Prefix Sections

The following is special processing that applies **only** to dsects built from the **XDCMAPS** SYM data and ADATA. These adjustments are **not** made for any maps loaded from any other source.

But for several standard IBM cblock maps, Z/XDC will make the necessary zero-point adjustments automatically when loading the dsect maps from the XDCMAPS module. (See the table below.)

Nicknames Support: To help make it more convenient to load and use maps having unexpected names and/or unexpected offsets, the **DMAP** command uses an internal, hardcoded, **nickname table** that allows a user to ask for any of the following maps by their nicknames.

NICKNAME	DSECT NAME	ZERO POINT ADJUSTMENT	NICKNAME	DSECT NAME	ZERO POINT ADJUSTMENT
ACB	IFGACB	none	NIB	ISTDNIB	none
BIND	ISTDBIND	none	RB	RBPRFX	-X’040’
CDE	CDENTRY	none	RPL	IFGRPL	none
CVT	CVTFIX	-X’100’	SCVT	SCVTSECT	none
DCB	IHADCB	none	SRB	SRBSECT	none
EXLST	IFGEXLST	none	TCB	TCBFIX	-X’020’
IOB	IOB	-X’010’	TCT	SMFTCT	none
JSCB	IEZJSCB	none			

Note that the table also allows the **DMAP** command to automatically adjust a map’s **zero point** to be the field that is commonly pointed to by other control blocks.

Nickname and Prefix Section Example:

DMAP XDCMAPS.TCB
DMAP XDCMAPS.TCBFIX
Both of these commands will load the dsect map named TCBFIX. But there **are** differences...

- The first command loads the map using its **nickname.**
- The second command loads the map using its **true name.**
- The first command will change the name of the dsect from TCBFIX to TCB, the second will not. (However, the name TCBFIX will remain available as a label within the TCB map.)
- Both of these commands will load either SYM data or ADATA depending upon whether or not the appropriate ADATA library (hlq.XDCV31.XDCADATA in this case) has been added to the currently active **MAPLIBS** list.
- The map created by the first command will have its zero point changed to the field named TCBRBP. The second command will leave map’s zero point at the map’s first field: TCBFRS.

Note, the DMAP command uses its nicknames support **only** for maps loaded from the

XDCMAPS load module or the XDCMAPSx ADATA files. The nickname support
is **not** available for maps loaded from any other load module or ADATA library.

XDCMAPS for Older z/OSs

IBM control blocks tend to be quite stable across z/OS releases, but many do change. So ColeSoft maintains at its website **(colesoft.com)** release specific collections of IBM control block mapping data for all z/OS releases going back to z/OS R1.6! See **HELP DUMPS MAPPINGDATA ZOSCBLOCKS** for more information.

The customers who are most likely to make use of this availability are those who use **dump/XDC** to examine **IPCS dumps** from third party systems. If such a system were running an older version of z/OS, then this historical repository will come in quite handy for them.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

ADATA - Accessing the ADATA form of XDCMAPS.
YOURDOC - Suggestions regarding creating a DOC module for your own control blocks.

Help MAPs Xdcmaps Adata

XDCMAPS is distributed in two forms:
- As a load module containing SYM data.
- As ADATA in a separate library.

SYM DATA

The load module form of XDCMAPS is named (strangely enough) **XDCMAPS.** It is distributed in the XDCLINK library. (The factory default name for that library is hlq.XDCV31.XDCLINK. Ask your Systems Programmer for the library's actual name at your Data Center.)

If the XDCLINK library is in your program's load module search order (task library, STEPLIB, JOBLIB, PLPA, or linklist), then dsect maps can be built from XDCMAPS SYM data just by issuing a suitable DMAP command. Example:

DMAP XDCMAPS.DCB

For more information, see HELP MAPS SYM.

If the XDCLINK library is not in your search order, then just add the library's name to the DMAP command. Example:

DMAP XDCMAPS.DCB hlq.XDCV31.XDCLINK

Note, this is necessary only for the debugging session's first reference to XDCMAPS. On subsequent references, the library's name can be omitted.

ADATA

The ADATA for XDCMAPS is located in the XDCADATA library in a series of members named XDCMAPSA, XDCMAPSB, ..., XDCMAPSZ. Each member contains the ADATA for those IBM control blocks whose **dsect** names start with the matching letter of the alphabet.

The XDCADATA library's factory default name is hlq.XDCV31.XDCADATA. Ask your Systems Programmer for its actual name at your Data Center.

Before you can use the ADATA form of XDCMAPS, you have to use the SET MAPLIBS command to make its library available to the DMAP command. Example:

SET MAPLIBS hlq.XDCV31.XDCADATA

For more information, see HELP MAPS ADATA MAPLIBS.

Then just use the DMAP command to build the desired dsect map. Example:

DMAP XDCMAPS.DCB

If you want to make the XDCADATA library a permanent default MAPLIB, then save it to your debugging session's profile using the MAPLIBS list named **DEFAULT.** This can be done with the following commands:

PROFILE READ
This is optional. It restores your currently active profile settings to your default values. (This is so that you don't accidentally permanently save other profile changes that you may have either forgotten about or not realized that you made.)

LIST MAPLIBS ALL
This displays all information both about the currently active MAPLIBS list and about all previously saved named MAPLIBS lists (if any).

SET MAPLIBS RESET
This is optional. It just clears the currently active MAPLIBS list (if any).

SET MAPLIBS hlq.XDCV31.XDCADATA
This adds XDCMAPS's ADATA library to the active MAPLIBS list.

SET MAPLIBS SAVE DEFAULT
This saves the currently active MAPLIBS list into the session's profile storage using the name **DEFAULT.** (Note, the name **DEFAULT** is the default name, it could have been left off of the command.)

PROFILE SAVE
This hardens the profile data by saving it to disk.

Note, because of shortcuts and defaults supported by Z/XDC, the above command stream can be shortened to the following:

```
PROFILE READ
SET MAPLIBS RESET hlq.XDCV31.XDCADATA SAVE
PROFILE SAVE
```

Help MAPs Xdcmaps Yourdoc

XDCMAPS, in addition to providing IBM control block maps, also serves as a good model for users to follow. If you are working with a system of programs consisting of numerous assemblies and numerous control blocks for passing data around amongst the components, then you might consider assembling and linkediting a "DOC" module that, like XDCMAPS, simply consists of the control blocks used by your programs. Doing this will make dsect maps of your control blocks easily accessible for use by Z/XDC.

If you are a JES2 programmer, then you might consider reassembling IBM's HASPDOC module with PARM=TEST and linkediting it into its own load module (also with PARM=TEST) into the JES2 load library. Then all of JES2's control block maps (the HCT, PCEs, JQEs, etc.) will be available for use by Z/XDC.

Help MAPs CIcsmaps

The module named XDCCICSM is distributed with the Z/XDC product and has no code in it; instead it provides both SYM data and ADATA for hundreds of CICS control block dsects. This module's data is available in the following places.

- **SYM data:** The SYM data version of the maps is located within a load module named XDCCICSM and that module is available in the library named hlq.XDCV31.XDCLINK.

- **ADATA:** The ADATA version of the maps is located in the library named hlq.XDCV31.XDCADATA under the XDCCICSM member.

You can use XDCCICSM as a quick and easy source for dsect maps of most common CICS control blocks. Just use Z/XDC's **DMAP** command to read any desired map from the module that contains the ADATA or SYM data for that map.

Notes

- "hlq.XDCV31.XDCxxxxx" are factory default names. Ask your Systems Programmer for the library's actual name at your Data Center.

- As with other IBM control blocks, a dsect often differs in name from the common name of the CICS control block which it describes. In general, the names of the CICS control block dsect maps are documented in the various "CICS Data Areas" manuals from IBM, and all share the prefix "DFH".

- The DFHCOMMAREA dsect appears twice, with different fields. XDCCICSM does not contain DFHCOMMAREA. Instead it has DFHCOMMAREA#1 and DFHCOMMAREA#2. The former details Web Interface Converter parms, while the latter details Web Interface Analyzer parms.

Example

The following example sets equates, loads maps, and places them in the Home Address Space such that the CICS/ESA Task Control Area of the current task can be formatted onto the screen.

DMAP XDCMAPS.TCB
This command will load the SYMDATA version of the map or the ADATA version, for the Task Control Block dsect, TCB, depending upon whether or not the hlq.XDCV31.XDCADATA library is in the currently active MAPLIBS list.

DMAP XDCMAPS.TCB SYMDATA
This coerces the ADATA/SYM data choice towards SYM data.

DMAP XDCMAPS.TCB ADATALIB=hlq.XDCV31.XDCADATA
This coerces the ADATA/SYM data choice towards ADATA, and it also explicitly tells z/XDC the name of the ADATA library, thus avoiding having to set up a MAPLIBS list. (But z/XDC under the covers will automatically add this library to the active MAPLIBS list so that it does not have to be specified a second time.) See HELP COMMANDS DMAP LOAD for more information.

LIST TASK
This command lists the TCBs in accessible address spaces. It also (re)generates a series of automatic equates, TCB#n. The following command uses the TCB#n of the **current task.**

USING TCB TCB#n
This command places the map onto the TCB that exists within the Home Address Space. In this example, we use the TCB#n of the **current task.**

DMAP XDCMAPS.TCBXTNT2
This command loads the ADATA map for the OS/VS1 - OS/VS2 Common Extension dsect, TCBXTNT2.

USING TCBXTNT2 TCB.TCBEXT2?
This command places the map onto the TCBXTNT2 that exists in the Home Address Space at the base address indicated by TCB's Common Extension Address field, TCBEXT2.

DMAP XDCCICSM.AFCB
This command loads the ADATA map for the CICS/MVS Authorized Function Control Block dsect, AFCB.

USING AFCB.DFHAFCB TCBXTNT2.TCBCAUF?
This command places the map onto the AFCB that exists in the Home Address Space at the base address indicated by TCBXTNT2's Subsystem Facility Control Block Address field, TCBCAUF.

DMAP XDCCICSM.DFHUKTCB RENAME=KTCB
This command loads the ADATA map for the User-defined Kernel TCB dsect, DFHUKTCB. For added convenience, DFHUKTCB is renamed to KTCB.

USING KTCB AFCB.AFCKTCB?
This command places the map onto the KTCB that exists in the Home Address Space at the base address indicated by AFCB's Kernel TCB Address field, AFCKTCB.

DMAP XDCCICSM.DFHUTASN RENAME=TAS
This command loads the map ADATA for the User Task Entry dsect, DFHUTASN, and for added convenience, renames DFHUTASN to TAS.

USING TAS KTCB.UKTCB_ACTIVE_TASK?
This command places the map onto the TAS that exists in the Home Address Space at the base address indicated by KTCB's Active Task field, UKTCB_ACTIVE_TASK.

DMAP XDCCICSM.DFHTCADS RENAME=TCA
This command loads the map ADATA for the CICS/ESA Task Control Area dsect, DFHTCADS, and renames it to TCA.

USING TCA TAS.UTATCAA?
This command places the map onto the TCA that exists in the Home Address Space at the base address indicated by TAS's TCA Address field, UTATCAA.

FORMAT TCA
Finally, this command displays the TCA from storage, formatted according to its ADATA map.

For related information select any of the following topics:
- HELP DEBUGGING CICS
- HELP MAPS SYM
- HELP MAPS ADATA
- HELP COMMANDS DMAP LOAD

Help MEssages

(For descriptions of all messages issued by z/XDC®,see the z/XDC® Messages Manual.)

Help MUltitask

If Z/XDC is being used to debug a multi-tasking program, it is quite possible for two or more tasks to ABEND (or reach breakpoints) and request debugging services at the same time. When this happens, Z/XDC will cause the second and all additional ABENDing tasks to wait either until the first task is resumed or terminated, or until you explicitly request that debugging of the first task be suspended in favor of debugging one of the other ABENDing tasks. (I call this "Conversation Management".)

Normally, when one task has control of debugging services, all other tasks are frozen out of those services until the first task, one way or another, releases control. The LIST TASKS command shows which tasks have ABENDed and which tasks are waiting pending control of the debugging conversation. The SWITCH command provides a mechanism by which you can switch control of the debugging conversation from one pending task to another. For more information, please see HELP COMMANDS SWITCH.

Please be aware that it is quite possible for a task to be within Z/XDC and waiting but not be waiting for control of the debugging conversation. This circumstance can occur through use of the TSO, ISPF, SPLIT, and SWAP commands. In these cases, Z/XDC loses control of the debugging conversation in favor of the processes started by these commands. Z/XDC then waits for the process to end before reacquiring control of the conversation. Thus if the process started by one of these commands should lead to Z/XDC gaining control in another task (e.g. TSO XDCCALL IEFBR14), a LIST TASKS command will NOT show the original task as pending conversation management. Instead, the task will only be awaiting completion of the TSO, ISPF, SPLIT, or SWAP

command. Further, the SWITCH command cannot be used to return control to the original task. Control will return to that task only when the TSO, ISPF, SPLIT, or SWAP command completes.

There are several other ways in which a task that has control of the debugging conversation can release that control:

- The task can resume execution and not return to debugging services for longer than a user settable timeout interval. (The factory default is 5 seconds. See HELP COMMANDS SET TIMEOUT for more information.)

- The task can terminate.

- The user can issue the SWITCH command to force the current task to relinquish control in favor of another task, (See HELP COMMANDS SWITCH for more information.)

When the current task releases control of the debugging conversation, other pending tasks compete for control of the conversation randomly (except when control is released via the SWITCH command).

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

SCOPE - Determining which tasks are within the scope of a debugging session.

CONVERSATIONMANAGEMENT - Managing display conflicts when both Z/XDC and the user program try to use the terminal at the same time.

IRBS - What happens when Z/XDC is interrupted and eventually a second Z/XDC becomes active under the same task.

Help MUltitask Scope

When Z/XDC receives control for the first time in an address space, it has to somehow determine which tasks will potentially participate in the debugging session and which tasks will not. This is because in a multitasking program it is quite possible that the first "call" to Z/XDC might occur in any subtask the programmer (or the vagaries of coding errors) might choose. Consequently, Z/XDC cannot simply assume that the first task under which it receives control is the debugging session’s top task.

Those tasks that are a part of the debugging session constitute the "scope" of the session. The scope of a debugging session can be displayed by the LIST TASKS command. Those tasks that are within the scope of a debugging session will be displayed hilighted. The top task in the scope of a debugging session is referred to as the "anchor task".

The "scope" of a debugging session has the following effects:

- All of Z/XDC's global control blocks are allocated from storage owned by the debugging session's anchor task.

- All tasks descendant from the anchor task constitute the scope of the debugging session.

- All other tasks are outside the scope of the debugging session.

- All maps, equates, breakpoints, etc. created by Z/XDC running under any task within the scope of the debugging session will be known to Z/XDC running under any other task falling within the scope of the same debugging session.

- When Z/XDC has received control and, therefore, has established the scope of its debugging session, then if Z/XDC should, for some reason, receive control in a task that is **not** within the scope of that first debugging session, then a new, independent debugging session is established, having its own scope and having its own, independent control block structure.

- If two, independent debugging sessions arise within an address space, then all maps, equates, breakpoints, etc. create within one of those sessions will be completely unknown to the other.

The creation of multiple debugging sessions within an address space can have the following consequences:

- A beneficial consequence is that independent and separate debugging sessions can be created within each SPLIT-screen within ISPF.

- A not so good consequence is that if a breakpoint known to one debugging session is placed into code that eventually will be executed within a task that is in the scope of another debugging session, then that other debugging session will only see and report an s0C1 ABEND. It will not know that it is a breakpoint.

- When debugging using the Cross Domain Facility, if a SWITCH command is used to switch control from the scope of one debugging session into the scope of another, then the debugging session will hang until an **ATTN** key is pressed.

Determining the Scope of a Debugging Session

To determine the scope of a debugging session, Z/XDC has to guess the future. It has to guess which current and future tasks might eventually pass control to Z/XDC and which will not. In order to make this guess, Z/XDC scans all of its ancestor tasks (running up the TCBOTC queue) looking for queued recovery routines (ESTAEs etc.) that have the potential of passing control to Z/XDC. The oldest task for which this potential exists becomes the debugging session's anchor task.

Scope Determination: SCB Backscan

To determine whether or not a task has a recovery routine that might pass control to Z/XDC, Z/XDC scans that task's SCB queue. "SCB"s are "STAE Control Blocks". They are queued from the TCBSTAB field. They define and describe the existence of recovery routines.

For each SCB, Z/XDC checks the following:
- Whether or not the exit address contained in the SCBEXIT field is the entry

point of the XDC load module.

- Whether or not the exit routine pointed to by SCBEXIT might branch and link to the XDC load module. (See HELP DEBUGGING ESTAES MODIFYING regarding writing an ESTAE that might call Z/XDC.)

Scope Determination: CALLSXDC String

Obviously, Z/XDC cannot possibly determine by simple inspection (or even complex inspection) whether or not a user written recovery routine might call Z/XDC. Instead, the programmer must create an explicit signal that Z/XDC can find. To do this, he must precede the recovery routine's entry address with the string "CALLSXDC". Example:

```
            DS    0H
            DC    CL8'CALLSXDC'
   MYESTAE  DS    0H
            ...
```

Scope Determination: //XDCJSTEP DD DUMMY

The above described process for scope determination is pretty good, but it is not perfect. After all, it's hard to predict the future. In particular, the scope determination process works fairly well for debugging sessions that run within a TSO address space, but when debugging within a background (batch) address space, they occasionally lead to problems. In TSO, the creation of multiple debugging sessions is sometimes what you might want, but in the background, it almost never is what you want.

To address this problem, you may add a **//xxxJSTEP DD DUMMY** card to your job's JCL. ("xxx" must match Z/XDC's current clone name, usually "XDC".) During scope determination, Z/XDC scans your job's TIOT, and if it sees this allocation, it will automatically make your job's jobstep task the anchor task for the debugging session. Generally, this eliminates the possibility of multiple debugging sessions occurring within the address space.

Within TSO, however, creating an xxxJSTEP allocation generally is **not** a good idea. This is because the jobstep task within a TSO address space usually is not a good candidate for anchoring the scope of a debugging session. This is because TSO commands and other programs that run within TSO do not run as the jobstep task, they run as descendant tasks. So if you've used Z/XDC to debug some command, and that command ended, the usefulness of that debugging session would end, but its life would not! Then if you tried to debug another command you would find a lot of trash lying around from the prior debugging session. So I don't recommend that you use the xxxJSTEP ddname within a TSO based debugging session.

Avoiding Multiple Debugging Sessions

To avoid creating multiple debugging sessions unintentionally, do any one (or more) of the following:

- If debugging a background (batch) program, add an //xxxJSTEP DD DUMMY card to your program's JCL.

- Use the XDCCALL program (or any of its aliases) to start the debugging session. Do this even if you already use ESTAEs (etc.) within your program for setting up

Z/XDC environments.

- issue an ESTAE for Z/XDC in your program’s main task.

- Code the "CALLSXDC" string ahead of the entry point of any main task ESTAE routine. Do this even if the ESTAE routine does not actually call Z/XDC. (It’s OK to lie.)

Help MUltitask Conversationmanagement

While debugging complex multitasking TSO or ISPF programs, it is possible for multiple tasks to ABEND and, therefore, to pass control to Z/XDC simultaneously. However, since there is only one terminal connected to the Home Address Space, Z/XDC has to manage its tasks so that no more that one task at a time attempts to communicate with the terminal. This is called "Conversation Management", and is explained in detail by the preceding topic (HELP MULTITASK).

TERMINAL CONFLICTS IN TSO

While Conversation Management is fully aware of all communications requests made by any Z/XDC running under any task, it is **not** generally aware of TPUTs and TGETs issued either by ISPF, by other user tasks or by system tasks. Although there are many techniques used by Conversation Management to minimize the resulting problems; nonetheless, it still is possible that Z/XDC will still issue TGETs in conflict with other tasks, or will fail to issue TGETs even when none are being issued by other tasks.

These problems arise basically because Z/XDC cannot always predict correctly when it is safe to use the terminal and when it is not. Normally, when a programmer writes a multi-tasking TSO or ISPF program, he takes care to make sure that only one task at a time attempts to use the terminal. Z/XDC, however, runs essentially asynchronously to the design of the user’s program: The user may wish to debug any task at any time without regard to whether or not it is his "communications" task or whether or not Z/XDC might be in conflict with the user’s communications task(s). Consequently, terminal control conflicts and keyboard lockouts are quite possible.

THE SOLUTION

If you are debugging in a scenario in which terminal control conflicts arise, then the best solution is to have Z/XDC communicate with a terminal different from the one owned by your TSO session. This can be done via cdf/XDC (see HELP XDCSRVER CDF).

(When in TSO) to trigger Z/XDC to attempt communication via cdf/XDC (instead of via its normal fullscreen displays), before starting your debugging session, you must allocate a dummy file to ddname //xxxCDF (where xxx is XDC’s current clone name). This can be done via the following TSO command: **ALLOC FILE(XDCCDF) DUMMY REUSE**

Then when you start your debugging session and Z/XDC receives control, it will not try to send any displays to your terminal. Instead, it will connect to cdf/XDC and wait for a programmer to signon to the debugging session. At this time you will be locked out of your terminal.

To proceed with your debugging session, you will have to go to a different terminal, logon to a different TSO session, go to Z/XDC's Startup Panel in ISPF (ask your Systems Programmer how), and select option **3** to connect to cdf/XDC. At this point (security permitting) you will see your original TSO session waiting for selection. When you select it, you will then be in a normal debugging session connected to your original TSO session.

Eventually, when you use Z/XDC's GO command to resume your program's execution, your Z/XDC terminal will lock up until your program either reaches a breakpoint, ABENDs, or terminates. If your program should happen to issues terminal displays, they will go to your original TSO session's terminal.

POSSIBLE RECOVERY

If for some reason you have decided not to use cdf/XDC and TSO terminal conflicts do occur, then recovery might be possible.

If a Z/XDC task and a user task are simultaneously displaying messages and requesting input, then the result will be a jumbled display, and the user will be unable to control which task will receive any commands that he might type. When this occurs, you should try Z/XDC's **WAIT command.** Although you can't control whether the Z/XDC task or the user task will receive this command, if you just continue typing it repeatedly, eventually, the Z/XDC task will get it and then will immediately cease trying to use the terminal. Instead, it will resume waiting in Conversation Management for control to be passed to it again. You then should be able to use the **PA1** or **PA2** key to restore the user task's display without further interference from the Z/XDC task. For more information, please see HELP COMMANDS WAIT.

In the event of a keyboard lockout, there is no task (Z/XDC or system or user) currently in communication with the terminal, and there is no process currently active that will lead to any task using the terminal. If Z/XDC is in control in one or more tasks, and if one or more of those Z/XDC's are waiting in conversation management for its turn to use the terminal, then it is possible to break the deadlock. To do so, you must signon to another TSO terminal, bring Z/XDC up (authorized or non-authorized, it doesn't matter), and issue the **THAW command** directed towards the deadlocked address space. If all of the above conditions are met, then any Z/XDC that is waiting in Conversation Management will be posted. This will cause it to wake up and send its display to the terminal. The keyboard will then unlock for input. For more information, see HELP COMMANDS THAW.

Help MUltitask Irbs

When Z/XDC is being used to debug a problem in a task, it is possible for an IRB to be queued to that task, thereby interrupting and freezing Z/XDC's execution at an arbitrary point. Furthermore, it is possible for the IRB routine itself to ABEND or reach a breakpoint, in which case a Z/XDC could be started for debugging the IRB. In this situation, Z/XDC is active twice under the same task with the second Z/XDC effectively interrupting the first, and with the first Z/XDC frozen at an arbitrary point and simply unable to proceed.

Here's an example:

- Suppose that your program has issued a STIMER macro and then shortly after that reached a breakpoint thereby giving control to Z/XDC.

- Now suppose that, while you are working with Z/XDC, the STIMER expires. This causes the System to add an IRB to the Request Block queue in order to run the STIMER exit routine. Thus, Z/XDC is interrupted at an arbitrary point in its processing.
- Now suppose that the STIMER exit routine itself ABENDs or reaches a breakpoint. This can cause Z/XDC to receive control again.

The problem is that when Z/XDC is interrupted at an arbitrary point during processing, its internal control blocks and data areas can be caught in an incomplete or undefined state. Now, if a second Z/XDC attempts to come in and share those control blocks, arbitrary damage and failures can occur.

To help avoid this kind of problem, when a new instance of Z/XDC unexpectedly interrupts an older instance of Z/XDC running under the same task, the newer Z/XDC is not allowed to share the older Z/XDC's data structures; therefore, it has to build new data structures that are separate from and independent of the older Z/XDC's structures.

Although the creation of temporary data structures solves the data corruption problem, it does lead to certain inconveniences:

- Because the interrupting Z/XDC and the interrupted Z/XDC have different data structures, all equates, maps, breakpoints and screen layouts established by the older Z/XDC will be unknown to the newer Z/XDC.

- In particular, if the newer Z/XDC received control due to a breakpoint set by the older Z/XDC, the newer Z/XDC will not know about the breakpoint. Instead, it will simply report that an s0C1 occurred, and you will have to manually restore the original opcode before you will be able to allow the IRB's execution to proceed.

- The new data structure being used by the newer Z/XDC is essentially temporary and is not suitable for use by Z/XDC's in other tasks. Consequently, the SWITCH command has to be disabled in order to prevent an Z/XDC in another task from (a) damaging the temporary data structures being used by the newer Z/XDC and (b) trying to use the older Z/XDC's data structures which are in an undefined state. So, you will not be able to use the SWITCH command until you finish debugging in the IRB routine, allowing the newer Z/XDC to terminate and allowing the older Z/XDC to resume.

Note, these kinds of problems do not always happen:

- If an IRB routine gains control while Z/XDC is running, but that routine does not itself pass control to Z/XDC, then no data corruption problem arises, and the interrupted Z/XDC will simply proceed as soon as the IRB routine completes. In fact, it is likely that you will never even be aware that the interrupt has occurred.

- If an IRB routine gains control in a task in which Z/XDC is **not** running, and if that IRB causes Z/XDC to receive control, then Z/XDC will process normally with its normal data structures. Preexisting equates, maps, breakpoints, etc. will be known and available to Z/XDC, and the SWITCH command also will be usable.

Help PRofiles

Z/XDC maintains a "profile" of information concerning your debugging session preferences and saves this information in your ISPF profile library (or other library) so that it will be available across debugging sessions.

Session profile data is maintained and used by Z/XDC **regardless** of whether your debugging session is running within or outside of ISPF. Although Z/XDC can and usually does use ISPF table and profile libraries to store its session profiles, it does **not** use ISPF services to access them. Instead, Z/XDC uses its own I/O routines. Accordingly, Z/XDC has access to its session profiles while running either within or outside of ISPF.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

- **MENU** - The Profile Menuing System.
- **MAPLIBLISTS** - Information about saving named lists of MAPLIB libraries (for source mapping) in the session profile.
- **READSAVE** - Reading, saving, resetting, and showing the status of session profiles.
- **NAMEDPROFILES** - Creating and using multiple, named profiles.
- **PROFNAMES** - Profile name formats and profile library member names.
- **DEFAULTPROFILE** - Default profile names and processing. Also, profile conversion information.
- **FACTORYDEFAULTS** - Hard-coded profiles loadable only by the **PROFILE RESET** command.
- **OLDPROFILES** - The conversion of profiles created by prior releases of Z/XDC.
- **DDNAMES** - The profile library search process.
- **TLIBPROFILES** - Creating named default profiles (and the standard default profile) in your site's ISPTLIB libraries.
- **WIDE** - Setting a default session profile that supports large geometry display terminals.

Help PRofiles MEnu

Z/XDC's **Profile Menuing System** can be used to display and set almost all of Z/XDC's profiled parameters (except **MAPLIB** lists). To activate the Menuing System, just use the **PROFILE** command without operands.

Selecting Submenus

Upon entering the Menuing System, the **Profile Home Menu** will be displayed showing a numbered index of submenus. You can select submenus in either of two ways:

- You can type **S** (select) in the Shortcut Entry Field (_) at the left side of a submenu's name. (With this method, you can select multiple submenus for display in succession.)
- And/or you can type the submenu's **index number** on the command line.

Saving Changes

Any changes that you make within the Menuing System will **not** be automatically saved to disk. If you want your changes to be permanently saved, then you have to either:

- Issue the **PROFILE SAVE** command after exiting the Profile Menuing System,
- Or type **YES** on the **Should this profile be saved now? ==>** question appearing on the Profile Menuing System's Home Menu.

The Home Menu

The Home Menu is the first menu that is displayed when you issue an unadorned **PROFILE** command. Its primary purpose is to be an index of the submenus where you can make changes to profiled settings.

You can select submenus in either of two ways:

- You can type **S** (select) in the Shortcut Entry Field (_) at the left side of a menu's name. (With this method, you can select multiple submenus for display in succession.)
- And/or you can type the submenu's index number on the command line.

The Home Menu also has:

- A place where you can set a description for the profile you are updating.
- A field where you can request the profile be saved to disk immediately.

The Profile Description

The Profile Menuing System's Home Menu contains a field in which you can type an arbitrary description for the profile. This description will then be upcased and:

- Saved by the **PROFILE SAVE** command,
- Displayed by the **LIST PROFILES** command.

The reason profiles have descriptions is because of the ability to save multiple, named profiles. A description can be used to remind you of why a particular profile exists.

If you wish to save an **unupcased** description, you can use the **SET PROFILE DESCRIPTION='string'** command.

The Submenus

Within, the submenus, you can display and change profiled parameters and values just by overtyping them.

PF keys

Within the **Profile Menuing System,** PF keys **PF3, PF4** and **PF5** can be used for navigation amongst (and out of) the menus. These PF keys are hard-coded, so their meanings cannot be changed within the Menuing System. They temporarily but unconditionally override whatever content you may have set via a prior **KEYS** command.

To remind you of this, all submenus have messages **(DBC722I)** at the top stating what the PF key settings are for that menu.

Your normal PFK settings will be restored when you exit The Menuing System.

The hardcoded PF keys are:

PF3
This ends the current submenu and takes you to the next higher menu in the Menuing System. If you are at the Home Menu, then **PF3** takes you out of the Menuing System back to Z/XDC's normal debugging session displays (with your normal PF keys restored).

Whenever you use PF3, any changes that you have made in the menu that you are exiting are placed into effect. (But, they are not saved permanently unless you later issue the **PROFILE SAVE** command.)

PF5
No matter where you are in the Menuing System, **PF5** takes you all the way out back to Z/XDC's normal debugging session displays. Any changes that you may have made are placed into effect. (But they are not saved permanently unless you later issue the **PROFILE SAVE** command.)

PF4
This discards all changes that you may have made on the **current** submenu and exits back to the Home Menu. Other submenus are not affected.

This PF key is **not** implemented in all submenus. When not implemented:
- Its **DBC722I** message will be **omitted** from the top of the submenu.
- Its use will fail.

All PF keys other than **PF3, PF4** and **PF5** retain whatever meanings that were assigned to them by a prior **KEYS** command. In particular, the various scrolling commands **(UP DOWN LEFT RIGHT)** can be used as one would normally expect.

Further PF key notes:

- The above settings for **PF3, PF4** and **PF5** are effective within the Profile Menuing System **regardless** of their settings in the rest of Z/XDC.
- The other major group of PF keys that can be used within the Menuing System are those that contain scrolling commands **(UP, DOWN, LEFT and RIGHT).**

- Generally, PF keys containing commands other that the scrolling commands will not be usable within the Profile Menuing System.

- Not all of the Menuing System panels support the **PF4** key. When this key is not supported, its use will result in an error message.

- Each Menuing System panel displays messages explaining what each supported PF key is and does.

- Except for scrolling commands, no other Z/XDC commands (not even the **END** command) are accepted from a Menuing System panel's command line.

Subtopics Index

The following are specific menu descriptions. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

01 **TSOISPFCDFCICS** - Describes the TSO/ISPF/CDF/CICS Settings menu
02 **TERMSETTINGS** - Describes the Terminal Specific Settings menu
03 **COLORS** - Describes the Terminal Colors menu
04 **PFKTOFKEYMAP** - Describes the PF Key to Function Key Assignments menu
05 **KEYS** - Describes the PF Keys Display and Update menu
06 **HKEYS** - Describes the Built-in Help PF Keys Display and Update menu
07 **LOG** - Describes the Session Logging Parameters menu
08 **WINDOWS** - Describes the Window Control Parameters menu
09 **READZAPPARSE** - Describes the READ ZAP and Parse Order Settings menu
10 **FORMATTRACEFIND** - Describes the FORMAT TRACE and FIND Settings menu
11 **AUTOMAPPRINTETC** - Describes the AUTOMAP PRINT and Other Settings menu
12 **REXX** - Describes the REXX Interface Settings menu
13 **CXDCSETTINGS** - Describes the c/XDC Settings menu
14 **USS** - Describes the USS interface settings menu Menu.

Help PRofiles MEnu TSoispfcdfcics

The "01 TSO/ISPF/CDF/CICS Related Settings" submenu allows you to specify:
- Your TSO fullscreen display method,
- The primary ISPF menu used by Z/XDC's **ISPF** command,
- Signon options for cdf/XDC,
- Whether or not Z/XDC would use CICS display services, if available.

Within this panel, you can change the following parameters by overtyping them.

- **Desired TSO communications interface**
  Selects which fullscreen display method to use when debugging in TSO.

TPUT - Uses fullscreen TPUTs to generate displays.
ISPF - Uses ISPF display services.

The default setting in all Factory Default profiles is **ISPF.**

The equivalent Z/XDC commands are:
SET TPUT
SET ISPF

- **Name of ISPF primary option menu**
Sets the initial panel used by ISPF when invoked via Z/XDC's **ISPF** command.

The default setting in all Factory Default profiles is **ISR@PRIM.**

The equivalent Z/XDC command is **SET ISR@PRIM.**

- **cdf/XDC**
The following cdf/XDC parameters relate to the time period during which cdf/XDC waits for a user to "signon" to a debugging session after a background batch job or system task has ABENDed and cdf/XDC has gained control.

  - **Let operators abort a debugging session's user signon wait**
  Controls whether or not cdf/XDC will display a WTOR message **(DBC640Q)** at the system's operator's console. ("WTOR" means "Write to Operator with Reply.")

  If this option is set to **YES,** cdf/XDC will display a WTOR giving the operator the opportunity to reply:
  C - Abort the waiting period and immediately percolate the pending ABEND.
  GO - Allow the user's program to resume execution without starting a debugging session. Please note that this may or may not work according to the nature of the ABEND and the program's logic.
  WTOR - Start the debugging session using the operator's console (instead of cdf/XDC) as the communications medium (via WTO/WTORs).

  - The default setting in all Factory Default profiles is **YES.**
  - The equivalent Z/XDC command is **SET DBC640Q.**
  - For more information, please see **HELP MESSAGES DBC640.**

  - **Signon wait time limit**
  Allows you to set the amount of time cdf/XDC should wait for a user to signon to a debugging session.

  Generally, when cdf/XDC gains control, it will only wait a limited amount of time for a user to signon. With this option, you can set the waiting time for signon ranging from 0 to 32767 minutes.

  The default setting in all Factory Default profiles is **60** minutes.

The equivalent Z/XDC command is **SET SIGNONWAIT.**

- **USERID to notify at signon time**
  Allows you to set whether or not cdf/XDC sends a message to a TSO session and (if so) which TSO session to sent it to. The message notifies the user that a debugging session is pending for him to connect to.

  If this option is left blank, cdf/XDC will not send any messages. On the other hand, if you type a TSO userid in the option field, cdf/XDC will send a message notifying that user when a program enters into a debugging session and needs the user to connect to that session.

The default setting in all Factory Default profiles is left **blank.**

The equivalent Z/XDC command is **SET USERID.**

For more information on logging on to cdf/XDC, see **HELP XDCSRVER CDF LOGON.**

- **CICS Options**
  This controls whether or not Z/XDC will use CICS display services when it finds itself running in a CICS transaction.

When you have finished changing parameters on this menu, you can press:
- **F3** to **accept** your changes and exit the current menu.
- **F4** to **reject** your changes and exit back to the Home menu.
- **F5** to **accept** your changes and exit the **entire** Profile Menu System.

If you've made a mistake and accepted changes when you didn't really mean to, you can still revert to the prechange settings by issuing a **PROFILE READ name** command to reload the unchanged profile from disk. This is because all the changes you may have made are not actually hardened until you issue a **PROFILE SAVE** command.

But once you do issue a **PROFILE SAVE [name]** command, all profile changes are hardened and can be reloaded in future debugging sessions. See **HELP COMMANDS PROFILE SAVE** for more information.

Help PRofiles MEnu TErmsettings

The **02 Terminal Settings** submenu allows you to display and change the following attributes for your 3270 terminal.

- **Terminal bell**
  Controls when your terminal's alarm (a beep usually) sounds.
  ERROR - The beep sounds only when error messages are displayed.

ALL - The beep always sounds when any message is displayed.
OFF - The beep never sounds.

The default setting in all Factory Default profiles is **ERROR.**

The equivalent Z/XDC command is **SET BELL.**

- **Screen size**
Controls your terminal's screen size.
STD - Sets the terminal's screen size to it's "primary" dimensions (24 lines by 80 columns - this, generally, is not desirable).
ALT - Sets the terminal's screen size to it's "secondary" dimensions (larger than 24 lines by 80 columns - this is always going to be what you want).

The default setting in all Factory Default profiles is **ALT.**

The equivalent Z/XDC commands are:
SET PRIMARYSIZE
SET SECONDARYSIZE

Generally, a terminal's secondary dimensions are defined by the workstation emulation program running on your PC. For more information about creating and changing such definitions, see **HELP FULLSCREEN TERMINALS GEOMETRIES.**

However, settings provided by the workstation program can be overridden by an improper **BIND image** defined in VTAM (running on the mainframe). For more information on this type of problem, see **HELP FULLSCREEN TERMINALS BINDIMAGES.**

- **fullscreen optimization status**
Controls whether Z/XDC attempts to optimize the screen image display process by writing to it only those fields that are different from the previously written screen image. The choices are:
ON - Z/XDC will attempt to optimize screen image traffic. (Optimization may not be possible in all TSO debugging environments.)
OFF - Z/XDC will not attempt to optimize screen image traffic. The screen will be erased and repainted every time a new display is generated. (This may cause an annoying flash,)
TRACE - Z/XDC will optimize most of its writes to the terminal; however writes occurring after a **TRACE** or **GO** command will not be optimized.

Note, these settings are effective only when Z/XDC is using its internal TPUT/TGET method for sending its displays to the terminal. When Z/XDC uses **ISPF Display Services,** display optimization is controlled by ISPF Display Services, and so it is always in effect.

The default setting in all Factory Default profiles is **ON.**

The equivalent Z/XDC command is **SET OPTIMIZATION.**

Notes regarding display optimization:

- Setting this option to **ON** can dramatically reduce the delay time between

pressing the ENTER key and receiving the regenerated display. This is especially true when your terminal is connected to your mainframe over low speed communication lines.

- If you have this option set to **ON** and for some reason your display becomes corrupted, simply press the **ATTN** key, the **PA1** key or the **PA2** key, and Z/XDC will fully repaint the display.

When you have finished changing parameters on this menu, you can press:
- **F3** to **accept** your changes and exit the current menu.
- **F4** to **reject** your changes and exit back to the Home menu.
- **F5** to **accept** your changes and exit the **entire** Profile Menu System.

If you've made a mistake and accepted changes when you didn't really mean to, you can still revert to the prechange settings by issuing a **PROFILE READ name** command to reload the unchanged profile from disk.

This is because all the changes you may have made are not actually hardened until you issue a **PROFILE SAVE** command.

But once you do issue a **PROFILE SAVE [name]** command, all profile changes are hardened and can be reloaded in future debugging sessions. See **HELP COMMANDS PROFILE SAVE** for more information.

Help PRofiles MEnu COlors

The **03 Color and Hilighting** submenu allows you set your display preferences for the input areas and protected fields for your terminal. This menu displays your current color, hilighting and intensity settings, and you can change them just by overtyping.

3270 Field Types

The 3270 terminal protocol supports four field types:

- **Low Intensity PROTECTED Fields**
  These fields are used to display ordinary static messages. They can't be changed, they can't be overtyped, they are just there to be seen.

- **High Intensity PROTECTED Fields**
  These fields are used to display special static messages such has title lines, section headers and anything that needs to be emphasized.

  Like all protected fields, these fields can't be changed, they can't be overtyped, they are just there to be seen.

- **Low Intensity INPUT Fields**
  These fields can be typed into. Z/XDC uses low intensity fields mostly for displaying data that can be zapped into and for form fill-in fields. Such fields may also be able to accept point-and-shoot commands and shortcut commands.

- **High Intensity INPUT Fields**
  These fields also can be typed into for zapping and other purposes, but Z/XDC uses high intensity when there is something additional and special about the field that needs to be noticed. Examples:
  - In register displays, when a register's content has changed, that content will be shown hilighted.
  - When Assembler machine instructions are displayed, the part (if any) of the instruction that accepts point-and-shoot commands will be shown hilighted.
  - Anything that you type into a command line will be shown hilighted.

Z/XDC allows you to distinguish these fields visually in several ways: By color, by intensity (bright vs. dim) and by extended hilighting (underlining, reverse video and even blinking [for the truly weird amongst us!]).

The **03 Color and Hilighting** submenu has the following sections:

Color Selections

The available display colors are **BLUE, MAGENTA, RED, YELLOW, CYAN, GREEN, WHITE,** and **DEFAULT.** The panel shows samples of these colors.

DEFAULT sets the colors to their hardware defaults: **GREEN, RED, BLUE** and **WHITE.**

Note, this is different from Z/XDC's **Factory** Default. Currently, that's **RED, WHITE, GREEN** and **YELLOW.**

Note, the color order used above is the same as the order in which the colors are listed in the menu's **Color Selections** section.

To change a color, just overtype at least the field's first character with the start of each color's name.

(For many people, the 3270's classic **BLUE** color is too dark. Z/XDC cannot do anything about that; **however,** your workstation emulator probably can! Most emulators (PCOMM, Vista, Attachmate, etc.) have dialogs for changing the RGB values it uses for the various 3270 color controls.)

Z/XDC's color settings can also be displayed and changed with the following commands:

- **LIST COLORS** reports the current settings.

- **SET COLORS abcd** lets you change the settings. **(abcd** are the first letters of your four color choices, given in the order shown by the **LIST COLORS** command.)

Extended Hilighting Selections

The available extended hilights are **REVERSE, UNDERSCORE, DEFAULT** and (god help you) **BLINK.**

DEFAULT sets the field to none of the above.

To set a hilight, overtype at least the first character of the field with the start of the hilight's name.

The hilight settings can also be displayed and changed with the following commands:

- **LIST HILIGHT** reports the current settings.
- **SET HILIGHT abcd** lets you change the settings. **(abcd** are the first letters of your four hilight choices, given in the order shown by the **LIST HILIGHT** command.)

Intensity Selections

The available intensities are **HIGH, LOW** and **DEFAULT.**

DEFAULT indicates that you intend to achieve hilighting via your color choices.

To set an intensity, you need to do two things:

1 - Indicate your intensity choices by overtyping at least the first character of a setting with either **H, L** or **D.**

2 - For those fields where you want your intensity choice to be effective, set the corresponding color choice to **DEFAULT.**

(INTENSITY is ignored for fields that have colors set.)

The intensity settings can also be displayed and changed with the following commands:

- **LIST INTENSITY** reports the current settings.
- **SET INTENSITY abcd** lets you change the settings. **(abcd** are the first letters of your four intensity choices, given in the order shown by the **LIST INTENSITY** command.)

When you have finished changing parameters on this menu, you can press:
- **F3** to **accept** your changes and exit the current menu.
- **F4** to **reject** your changes and exit back to the Home menu.
- **F5** to **accept** your changes and exit the **entire** Profile Menu System.

- If you've made a mistake and accepted changes when you didn't really mean to, you can still revert to the prechange settings by issuing a **PROFILE READ name** command to reload the unchanged profile from disk.
- This is because all the changes you may have made are not actually hardened until you issue a **PROFILE SAVE** command.
- But once you do issue a **PROFILE SAVE [name]** command, all profile changes are hardened and can be reloaded in future debugging sessions. See **HELP COMMANDS PROFILE SAVE** for more information.

Help PRofiles MEnu Pfktofkeymap

The **04 PF Key to Function Key Assignments** submenu allows you to view and update the function key ranges to which each of your PF key sets is assigned. This menu also provides a set of rules to follow when you are assigning PF key sets to function key ranges.

Basic Definitions

A **PF key set** is a set of 12 program function key command definitions.

A **function key** is a physical key on your terminal's keyboard.

A **function key range** is a group of 12 function keys.

Most terminals have 12 function keys capable of transmitting to the mainframe a total of 24 PF key order codes.

Some older terminals have only 10 function keys capable of transmitting up to 20 PF key order codes.

However, the 3270 protocols permit up to **36** PF key order codes, so Z/XDC supports terminals having up to 36 function keys. (Historically, there have been at least two 3270 Terminal Emulations capable of transmitting all 36 PF key codes.)

Mapping Z/XDC PF Keys to Terminal Function Keys

Z/XDC organizes the PF key command definitions into three sets: SET-A, SET-B and SET-C. Each set has 12 individual PF key definitions.

With the **04 PF Key to Function Key Assignments** submenu, you can arrange Z/XDC's three PF key sets so that they map to whatever function key ranges your terminal is capable of transmitting, whether it is:

    10 by 1
    10 by 2
    12 by 1

12 by 2
12 by 3
("12 by 2", for example, means twelve function keys capable of transmitting 2 ranges of PF key order codes [usually with the help of the SHIFT key.]

For more information on PF key sets, see **HELP FULLSCREEN PFKEYS PFKEYSETS.**

To change a PF key set's assignment in this menu, overtype the existing set (A, B, C or blank) to the right of the PF key range in the column that corresponds to your terminal type. Note, for terminals with only 10 function keys, this menu allows you to choose whether your second key set starts with PF key #11 or PF key #13.

For suggestions and examples of assigning PF key sets to function key ranges for specific terminals, see **HELP FULLSCREEN PFKEYS PFKEYSETS SUGGESTIONS.**

For information on using the **SET KEYS** command to assign PF keys sets to ranges of function keys, see **HELP COMMANDS SET KEYS.**

Other Controls Available on the 04 PF Key to Function Key Assignments Menu

Near the bottom of the menu, you can use the following option to access the PF key Display and Update Menu. With this menu, you can assign command strings to individual PF keys.

- Select here for the **05 Function Keys**
  Type **S** (Select) in the entry field at the left of this option to go to the PF keys Display and Update Menu.

The corresponding line mode command for many of this menu's capabilities is **SET KEYS.**

When you have finished changing parameters on this menu, you can press:
- **F3** to **accept** your changes and exit the current menu.
- **F5** to **accept** your changes and exit the **entire** Profile Menu System.
- F4 is not yet supported in this menu.

If you've made a mistake and accepted changes when you didn't really mean to, you can still revert to the prechange settings by issuing a **PROFILE READ name** command to reload the unchanged profile from disk. This is because all the changes you may have made are not actually hardened until you issue a **PROFILE SAVE** command.

But once you do issue a **PROFILE SAVE [name]** command, all profile changes are hardened and can be reloaded in future debugging sessions. See **HELP COMMANDS PROFILE SAVE** for more information.

Help PRofiles MEnu Keys

The **05 Function Keys** submenu allows you to view and assign command strings for any

or all of your PF keys. Within this menu, you can change your individual PF key command definitions simply by overtyping them in the display. Also, you can cycle through the PF key sets assignments (SET-A, SET-B or SET-C) by pressing the ENTER key.

Note, when typing in your PF key command definitions, keep in mind that Z/XDC's processing of PF keys provides for the automatic or manual inclusion of variable information before executing the PF key's commands. For detailed information, see a series of topics starting with **HELP FULLSCREEN PFKEYS.**

Other Controls Available on the 05 Function Keys Menu

At the bottom of this menu, there are a couple of selections that will take you directly to other, related menus:

- The **06 Function Keys for Help Topics** submenu allows you to change the PF key definitions that are active during **Help Topic** displays.
- The **04 PF Key to Function Key Assignments** submenu allows you to chose which PF key 12-key ranges will be assigned to which of your PC's 12-key function key ranges.

You may jump to these other submenus either:

- By typing an **S** into their shortcut fields,
- Or by typing their menu number on the command line.

Note, outside of the Profile Menuing System, you can shortcut directly to the PF keys Display and Update Menu by using the **KEYS** command from Z/XDC's normal command line. You do not have to wade through the Profile Menuing System to get here.

The corresponding Z/XDC command for many of this menu's capabilities is **SET KEYS** .It allows you:

- To assign command strings to individual PF keys.
- To assign PF key sets to function key ranges.

When you have finished changing parameters on this menu, you can press:

- **F3** to **accept** your changes and exit the current menu.
- **F5** to **accept** your changes and exit the **entire** Profile Menu System.
- F4 is not yet supported in this menu.

If you've made a mistake and accepted changes when you didn't really mean to, you can still revert to the prechange settings by issuing a **PROFILE READ name** command to reload the unchanged profile from disk. This is because all the changes you may have made are not actually hardened until you issue a **PROFILE SAVE** command.

But once you do issue a **PROFILE SAVE [name]** command, all profile changes are hardened and can be reloaded in future debugging sessions. See **HELP COMMANDS PROFILE SAVE** for more information.

Help PRofiles MEnu Hkeys

The **06 Function Keys for Help Topics** submenu displays two things:

- It displays (and allows you to change) the PK key command strings that are activated when The **HELP** command is used to display **Help topics.**
- It displays (and allows you to change) the PF key **nicknames** that are displayed at the bottom of the topics display.

Z/XDC maintains a separate set of PF key definitions for use only when you are displaying a **Help topic.** This is done, of course, because the PF key definitions that are suitable for debugging sessions are not at all appropriate for navigating the Built-in Help.

For more information about the default **Help mode PF key** definitions, see **HELP FULLSCREEN PFKEYS HELPKEYS.**

By default, Z/XDC shows nicknames of the **Help Mode PF key** definitions at the bottom of your screen when you are viewing the Built-in Help. This is done both to remind you that the PF key definitions are different, and to show you hints of what they are.

Help Mode PF Key Display

The **06 Function Keys for Help Topics** submenu allows you to view and change the command strings for your **Help Mode PF keys.** Within this menu, you can change individual **Help Mode PF keys** simply by overtyping them in the display to the right of the PF key number (1st through 12th).

Help PF Key Hints Display

At the bottom of the Built-in Help Keys Menu, you can also change any or all of the individual PF key nicknames by overtyping the displayed text. These nicknames are intended to remind you that the **Help Mode PF key** definitions are enabled when you are viewing the Built-in Help and to hint at the current definition for each of the keys.

Other Controls

Near the top of the submenu, you can use the following options to enable or disable the **Help Mode PF keys** and to control whether Z/XDC displays the **Help Mode PF key** nicknames when viewing the Built-in Help.

- **Are these PF key settings active during Built-in Help?**
  The default setting in all Factory Default profiles is **YES.**

- **Should the PF key's nicknames be displayed during Built-in Help?**
  The default setting in all Factory Default profiles is **YES.**

Note, outside of the Profile Menuing System, you can shortcut directly to the Built-in Help Keys Menu by using the **HKEYS** command from Z/XDC's normal command line. You do not have to wade through the Profile Menuing System to get here.

The corresponding Z/XDC command for many of this menu's capabilities is **SET HKEYS.** It allows you:
- To assign command strings to individual **Help Mode PF keys.**
- Control whether or not the **Help Mode PF keys** are to be active during Built-in Help displays.
- Control whether or not the **Help Mode PF key** nicknames are displayed during Built-in Help Displays.

Note, there is no Z/XDC command available for setting/changing **Help Mode PF key nicknames.**

When you have finished changing parameters on this menu, you can press:
- **F3** to **accept** your changes and exit the current menu.
- **F5** to **accept** your changes and exit the **entire** Profile Menu System.
- F4 is not yet supported in this menu.

If you've made a mistake and accepted changes when you didn't really mean to, you can still revert to the prechange settings by issuing a **PROFILE READ name** command to reload the unchanged profile from disk. This is because all the changes you may have made are not actually hardened until you issue a **PROFILE SAVE** command.

But once you do issue a **PROFILE SAVE [name]** command, all profile changes are hardened and can be reloaded in future debugging sessions. See **HELP COMMANDS PROFILE SAVE** for more information.

Help PRofiles MEnu Log

The **07 Session Logging Parameters** menu allows you to display and change settings related to session logging.

Z/XDC supports writing a session log either to disk (a normal sequential file) or to spool (queued either for "printing" or for retrieval from a TSO session). This log consists of copies of all of the commands issued from your session's primary display window, as well as the displays, reports, and actions produced by those commands.

Some log control settings pertain only to logs written to spool, others pertain only to logs written to disk, and some pertain to both.

General Log Control Settings

- **Automatic session logging in effect?**
  A scrollable log of your debugging session is always generated in storage. This

setting controls whether that log is written to disk or spool either automatically or manually. When manually, the log is not written out unless and until a **LOG** command is issued. So the settings for this field are:

YES - Log records are automatically written to disk/spool immediately as they are generated.

NO - Log records are generated, but they are not written out unless and until a **LOG** command is issued.

The default setting in all Factory Default profiles is **YES.**

The equivalent Z/XDC commands are:

SET LOG AUTO

SET LOG MANUAL

- **Number of scrollable messages**

As you issue commands and Z/XDC generates responses, a scrollable log of your activity is built in storage. This parameter allows you to set the size (number of log records) of that in-storage copy of the log. You may enter a decimal number ranging from **100** to **65535.**

The default setting in all Factory Default profiles is **65,535** records.

The equivalent Z/XDC command is **SET LOG SIZE=nnnnn.**

Note, when manual logging is in effect, if the in-storage log scrolling area fills up before a **LOG** command is issued, then the oldest log records are discarded and lost.

- **Write log files to disk or spool?**

When logging occurs, this control where the log is written to. The choices are:

DISK - Log records are written to a sequential file on DASD.

SPOOL - Log records are written to a "printer" file on spool.

The advantages and disadvantages of these choices are described in detail in **HELP FULLSCREEN LOGGING.**

The default setting in all Factory Default profiles is **SPOOL.**

The DISK/SPOOL setting causes those parts of the **07 Session Logging Parameters** menu to be **hilighted** that pertain to the chosen option.

There is no explicit equivalent Z/XDC command for the DISK vs. SPOOL setting. Instead, the setting is inferred from the use of other **SET LOG** command operands that relate either to SPOOL logging or DISK logging. For more information, see **HELP COMMANDS SET LOG MANAGEMENT.**

Spool Related Log Settings

These settings become active when the log is to be written to spool.

- **Dataset SYSOUT class**

This specifies the SYSOUT class to which the log is to be written. Permitted

values are, of course, A-Z and 0-9. However, you should choose a class:
- That is a HOLD class.
- That is not a dummy class.
- That is designated by your Data Center as being intended for TSO users.
- That is probably not ever going to be actually printed (or punched).

The default setting in all Factory Default profiles is SYSOUT class **X.**

The equivalent Z/XDC command is **SET LOG SYSLOG.**

- **Place in Held status?**
This overrides the selected class's HOLD attribute. The choices are:
YES - The log file is forced into HOLD status regardless of whether or not the SYSOUT class is a HOLD class. (This is the highly recommended setting to use.)
NO - The log file is forced into NON-HOLD status.

The default setting in all Factory Default profiles is **YES.**

The equivalent Z/XDC command is **SET LOG HOLD.**

- **Printer destination**
If you actually want your session logs to be printed from time to time, then you can specify the name of the printer here. Ask your Systems Programmer for the name of the printer that you want to use.

The default setting in all Factory Default profiles leaves this value **blank.**

The equivalent Z/XDC command is **SET LOG DESTINATION.**

- **Output Limit (OUTLIM)**
This sets a number-of-lines limit that JES2 is to enforce. The value may be:
0 - No specific output limit is assigned to the log file; however, if an output limit has been defined for the job, then all output written to the log will count against that limit.
number - (A decimal number ranging from **1,000** to **16,777,215)** The output limit for the log file is set to this value, and the log file's output does not count against the job's limit.

This setting generally would be used by customers at data centers that have excessively low system default limits. If you have to use this setting, then I guess I would recommend setting it to 50,000 or so.

The default setting in all Factory Default profiles is **0** (no special output limit). The equivalent Z/XDC command is **SET LOG OUTLIM.**

Disk Related Log Settings

These settings become active when the log is to be written to a file on disk (not

spool).

- **Unparsed DSNAME**

This specifies the dsname that Z/XDC is to use when creating the log file. "Dsname" must be a fully qualified, UNquoted dataset name. It may contain variable symbols for which Z/XDC will make substitutions. This allows the dsname to be reactive to the current time, date, userid, jobname, etc. For syntax details, especially regarding the variable symbols, see **HELP COMMANDS SYNTAX DSNAMES.**

Z/XDC will create log files when necessary and will use existing files when present.

The default dsname in all Factory Default profiles is ***P.*XLOG.*D.*T.** The menu will show a sample resolution of whatever the dsname setting is.

The equivalent Z/XDC command is **SET LOG DSNAME.**

- **Allocation volume**

If If you would like to force log datasets to be created on a particular DASD volume, you can provide the volser here.

The default setting in all Factory Default profiles leaves this value **blank.** This allows the Operating System to choose the volume.

The equivalent Z/XDC command is **SET LOG VOLSER.**

- **Allocation unitname**

If you would like to force log datasets to be allocated to a particular group of DASD volumes, you can provide the appropriate unit name here.

The default setting in all Factory Default profiles leaves this value **blank.** This allows the Operating System to choose the group of volumes to consider.

The equivalent Z/XDC command is **SET LOG UNITNAME.**

- **Append or overwrite?**

This controls what Z/XDC does if it finds that a particular log dataset already exists. The choices are:

APPEND - New log data will be appended to the end of the existing dataset.

OVERWRITE - New log data will be written beginning at the start of the dataset. Thus, old log data (if any) will be lost.

The default setting in all Factory Default profiles is **APPEND.**

The equivalent Z/XDC command is **SET LOG DISPOSITION.**

When you have finished changing parameters on this menu, you can press:
- **F3** to **accept** your changes and exit the current menu.
- **F4** to **reject** your changes and exit back to the Home menu.
- **F5** to **accept** your changes and exit the **entire** Profile Menu System.

If you've made a mistake and accepted changes when you didn't really mean to, you can still revert to the prechange settings by issuing a **PROFILE READ name** command to reload the unchanged profile from disk. This is because all the changes you may have made are not actually hardened until you issue a **PROFILE SAVE** command.

But once you do issue a **PROFILE SAVE [name]** command, all profile changes are hardened and can be reloaded in future debugging sessions. See **HELP COMMANDS PROFILE SAVE** for more information.

Help PRofiles MEnu Windows

Z/XDC displays always contain one or more command lines. (The default setting in all Factory Default profiles is **two** command lines.) Each command line starts an area of the display called a **window.** The results of all Z/XDC commands issued on a command line are displayed on the display lines immediately following the command line.

The **08 Window Controls** submenu allows you to display and change settings related to any command line and its window. This submenu will have as many selections as you have windows defined in your debugging session. Selecting any one of them will take you to a **WINDOW OPTIONS AND ATTRIBUTES** submenu from which you can view and change the following:

Default vertical scroll
This is the default distance the display will advance in response to a bare **UP** or **DOWN** command. The choices are **CURSOR, HALF, DATA, FULL,** and a specific **number of lines.**

The default setting in all Factory Default profiles is **CURSOR.**

The equivalent Z/XDC command is **SET WINDOW VERTICAL.**

Default horizontal scroll
This is the default distance the display will shift sideways in response to a bare **LEFT** or **RIGHT** command. The choices are **CURSOR, HALF, DATA, FULL, MAX,** and a specific **number of columns.**

The default setting in all Factory Default profiles is **30** columns.

The equivalent Z/XDC command is **SET WINDOW HORIZONTAL.**

Autoreset horizontal scroll?
This controls whether or not a horizontal scroll will be canceled for a window each time a new non-scrolling command is issued. The permitted values are:

YES - If the window's display has been shifted sideways, the next time a non-scrolling command is issued for that window, the display will be restored fully shifted back to the left.
NO - A window's left/right shift persists from one command to the next.

The default setting in all Factory Default profiles is **NO** for the primary Primary Window and **YES** for all secondary display windows.

The equivalent Z/XDC command is **SET WINDOW HORIZONTAL.**

Recallable commands list size
For each window, Z/XDC maintains a commands recall list from which previously issued commands can be retrieved to the command line. This field controls the maximum number of commands that can be saved for possible retrieval. Permitted settings range from 1 to 255.

The default setting in all Factory Default profiles is **255** commands.

The equivalent Z/XDC command is **SET WINDOW RETRIEVE.**

Although the **WINDOW OPTIONS AND ATTRIBUTES** submenu allows you to change options for any window, it does not allow you to create or delete windows. That can be done only by these commands:
- **SET WINDOW CREATE**
- **SET WINDOW DELETE**

When you have finished changing parameters on this menu, you can press:
- **F3** to **accept** your changes and exit the current menu.
- **F5** to **accept** your changes and exit the **entire** Profile Menu System.
- F4 is not yet supported in this menu.

If you've made a mistake and accepted changes when you didn't really mean to, you can still revert to the prechange settings by issuing a **PROFILE READ name** command to reload the unchanged profile from disk. This is because all the changes you may have made are not actually hardened until you issue a **PROFILE SAVE** command.

But once you do issue a **PROFILE SAVE [name]** command, all profile changes are hardened and can be reloaded in future debugging sessions. See **HELP COMMANDS PROFILE SAVE** for more information.

Help PRofiles MEnu REAdzapparse

The **09 READ ZAP and Parse Order Settings** submenu allows you to display and change all of the settings relating to the **READ** command, the **ZAP** command and the **Parsing Order.**

Specifically, the following settings can be displayed and changed. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP**

***FORWARD (F5)** to skip.

STOREPROTZAP - Allowing the zapping of store protected storage
MNEMONICZAP - Limiting mnemonic zaps according to instruction length
READECHO - Echoing command scripts in real time as they are processed
SEQFIELD - Managing the detection of sequence number fields.
SCRIPTLIBNAME - Specifying a default Scripts Library dsname
PARSEORDER - Defining a language parsing order for High level Language storage referencing expressions

When you have finished changing parameters on the **09 READ ZAP and Parse Order Settings** submenu you can press:
- **F3** to **accept** your changes and exit the current menu.
- **F4** to **reject** your changes and exit back to the Home menu.
- **F5** to **accept** your changes and exit the **entire** Profile Menu System.

If you've made a mistake and accepted changes when you didn't really mean to, you can still revert to the prechange settings by issuing a **PROFILE READ name** command to reload the unchanged profile from disk. This is because all the changes you may have made are not actually hardened until you issue a **PROFILE SAVE** command.

But once you do issue a **PROFILE SAVE [name]** command, all profile changes are hardened and can be reloaded in future debugging sessions. See **HELP COMMANDS PROFILE SAVE** for more information.

Help PRofiles MEnu REAdzapparse SToreprotzap

The "09 READ ZAP and Parse Order Settings" submenu has a setting captioned **Allow Zapping of Store Protected Storage?**

When Z/XDC is running authorized, its **ZAP** command can be used to store data into nearly any area of storage **including store protected** storage! But this **Allow Zapping of Store Protected Storage** option can be used to limit Z/XDC's ability to store into store protected storage.

The choices are:

YES - Allows the **ZAP** command to store into store protected storage (security and authority permitting, of course).

NO - Prevents Z/XDC from being able to store into store protected storage even when both authority and security would permit it.

The default setting in all Factory Default profiles is **NO.**

The equivalent Z/XDC command is **SET ZAP NORMAL|SPROT.**

Note that the **SET ZAP NORMAL** setting is **NOT(!)** a security feature. It is an **integrity** feature. It exists simply to help users keep themselves from making common

mistakes.

For more information on security issues regarding the **ZAP** command, see **HELP SECURITY ZAP.**

Help PRofiles MEnu REAdzapparse Mnemoniczap

The "09 READ ZAP and Parse Order Settings" submenu has a setting captioned **Limit Mnemonic Zaps by Instruction Length?**

Z/XDC's fullscreen display interface has many convenience features. One of these is the ability to zap displayed machine instructions simply by overtyping:
- Either the opcodes displayed in hex
- Or the instruction mnemonics displayed as text
- (Or even both)

When performing such zaps, it is quite possible for the user to change an instruction from one having a particular length (2 4 or 6 bytes) to one having a different length. There are two ways he can do this:
- One is by overtyping the hexadecimal opcode in the displayed machine code.
- The other is by overtyping the displayed mnemonic.

But of course zapping an instruction into another having a different length can lead to problems unless you really know what you are doing. (See below.)

This setting **partially** controls whether or not Z/XDC will allow the user to change an instruction to one having a different length. Specifically, the choices for this setting are:

YES - If the user changes the machine instruction by overtyping the displayed **mnemonic,** an Instruction Length Check (ILC) **is** performed, and the change will be disallowed if the new machine instruction has a length different from the current one.

On the other hand, if the user changes the machine instruction by overtyping its displayed **hex,** an Instruction Length Check is **not** performed, and the change will be allowed to occur.

This, of course, can result in some **seriously broken code,** so you really need to know what you're doing before trying this!

NO - The Instruction Length Check will **not** be performed regardless of what is overtyped, and the change will be allowed to occur.

This, again, can result in some **seriously broken code,** so be careful!

The default setting in all Factory Default profiles is **YES** (perform the length check).

The equivalent Z/XDC command is **SET ILC/NOILC.**

Notice: When you perform a zap that changes instruction lengths, surrounding code

is **not** shifted to make room for the new instruction. Instead, the interpretation of subsequent instructions will become desynchronized from actual instruction boundaries, and that usually is a very bad thing!

So if you're going to use zaps that change instructions regardless of their lengths, you had better know what you're doing.

Help PRofiles MEnu REAdzapparse Readecho

The "09 READ ZAP and Parse Order Settings" submenu has a setting captioned **Echo Scripted Cmds and Displays as They Happen?**

Normally, when processing a **READ** command to read and execute a script, Z/XDC processes each command in the script and internally generates displays as it goes along. However, because of the nature of fullscreen display management, the resulting messages are not displayed until the script completes.

If the script is long, this can result in a rather lengthy blank-screen/keyboard-lockout time. So this setting can be used to change that behavior. The choices are:

YES - All messages generated by a script are displayed (temporarily) as they are generated. But these messages are displayed using a line mode protocol, so they appear outside of Z/XDC's fullscreen display management. But when the script completes, the messages are erased and Z/XDC's normal fullscreen display is restored. The messages then reappear within the fullscreen display.

NO - A temporary display of the generated messages is not produced, and the keyboard remains locked, and the display remains frozen until the script completes.

The default setting in all Factory Default profiles is **YES.**

The equivalent Z/XDC command is **SET READ ECHO=YES|NO.**

Note, **YES** is effective only when debugging programs that are running within TSO. It is **not** effective when debugging programs **(via cdf/XDC)** that are running in the background. In this case, the keyboard always remains locked, and the display remains frozen until the script completes.

Help PRofiles MEnu REAdzapparse SEqfield

The "09 READ ZAP and Parse Order Settings" submenu has a setting captioned **Script File Sequence# Field?** This controls how Z/XDC determines whether or not a script file has a sequence# field.

When processing a script, the **READ** command needs to ignore sequence fields when present. So it needs to know **if** they are present. This setting controls whether or not Z/XDC will try to automatically detect the presence or absence of sequence number fields. The choices are:

PRESENT - This informs Z/XDC that sequence number fields are to be consider present, and he should ignore their content regardless of whether or not that actually contain numbers.

ABSENT - This informs Z/XDC that sequence number fields are to be consider absent, and he should process their content as if they were a normal part of the rest of each record's data.

DETECT - This informs Z/XDC that he is to attempt to detect for himself the presence or absence of sequence number fields and proceed accordingly.

These days, sequence number fields might contain actual sequence numbers, but then again they might not. Consequently, the rules for detecting the presence of a sequence number field are far from straight forward. If you want the gory details, you can find them at **HELP SCRIPTS RYOSCRIPTS SEQUENCEFIELDS.**

The default setting in all Factory Default profiles is **DETECT.**

The equivalent Z/XDC command is **SET READ SEQUENCEFIELD=YES|NO.**

Help PRofiles MEnu REAdzapparse SCriptlibname

The "09 READ ZAP and Parse Order Settings" submenu has a setting captioned **Default Scripts Library Name.**

This lets you define an initial library to be used by the **READ** command for finding an running Z/XDC command scripts.

The **READ** command can read Z/XDC command scripts from any sequential file, be it a sequential dataset or a member of a PDS or PDSE. But generally it is most convenient for scripts to be gathered together into a PDS[E] (a library).

To make the best advantage of the convenience of a library, this option can be used to define (or clear) a default Scripts Library name. The **READ** command can then read members of this library simply by providing just the member name.

To clear the default library name, simply blank out the library name from the input field. (Thereafter, the field will show **(none)** until you type in a new library name.)

Notice, this setting is only an **initial** default library name. When you run a script you may:

- Name either a member of this library (Example: **READ TSOMAPS)**

- Or you may name a member of an entirely different library (Example: **READ**

FRED.XDCSCPTS(SCRIPT1)

But notice! When you do this, the given library name becomes the new default. So, for example: **READ SCRIPT2** will read SCRIPT2 from FRED.XDCSCPTS, not from the library previously shown by this submenu.

Further, this submenu will, going forward, show FRED.XDCSCPTS as the **new default** script library.

The default setting in all Factory Default profiles is **hlq.XDCV31.XDCCMDS.**

The equivalent Z/XDC command is **SET READ DSN=dsname.**

An alternate way to set a default script library name is to use the library name on a **READ** command. Example: **READ DAVE.SCRIPTS(MYSCRIPT)** both runs the **MYSCRIPT** script and sets **DAVE.SCRIPTS** as the new default scripts library name.

See **HELP COMMANDS SET READ** for detailed information regarding requirements for Script Libraries.

Help PRofiles MEnu REAdzapparse Parseorder

The "09 READ ZAP and Parse Order Settings" submenu has several settings captioned **nth Language in the Parse Order.**

These control the order in which language parsers are given a chance to parse various Assembler symbols and C/C++ variables.

Languages Parse Order

The syntax rules for variables differ from one language to another such that a syntax that is valid in one language is invalid in another. Therefore, in order to support debugging in multiple languages, Z/XDC must use different parsers from one language to the next.

However, in order to integrate as much as possible multiple languages support, we want Z/XDC to accept the syntaxes of a variable regardless of its language. So when Z/XDC parses a variable, it will attempt to parse it first according to one language, but if that fails, then according to another language, and so one. Only if a variable fails to parse in all languages with Z/XDC declare a syntax error.

To that end, the "09 READ ZAP and Parse Order Settings" submenu has multiple **nth Language in the Parse Order** fields that allow you to define:

- (a) Which available parsers are going to be used when parsing a variable,
- (b) The order in which the various parsers are attempted,
- (c) Which parser is going to generate the Syntax Error message should all parsers fail. (That will be the **last** parser listed.)

When all parsers fail, a syntax error message needs to be produced. But of course the nature of the syntax error will vary according to the intended parsing language,

so the question arises, which language parser should build the error message?

Well as just noted, we have decided to make the last listed parser be the one responsible for the error message. But usually you might prefer the first parser be the one responsible for the error message. Well, that's simple... just provide the first parser's name again, making it also the last parser.

Example:

CEE 1st Language in the Parse Order
ASM 2nd Language in the Parse Order
CEE 3rd Language in the Parse Order

For more information, see **HELP ADDRESSING PARSERS.**

The default settings are as follows:

Factory Default Profile	**Setting**
AWIDE	ASM then CEE
A80	ASM then CEE
CWIDE	CEE then ASM
C80	CEE then ASM

The equivalent Z/XDC command is **SET PARSEORDER.**

Help PRofiles MEnu Formattracefind

The **10 FORMAT TRACE and FIND Settings** submenu allows you to display and change a variety of settings for several Z/XDC commands.

(For changing settings regarding the **READ** and **ZAP** commands, see **HELP PROFILES MENU READZAPPARSE.)**

Subtopics Index

The following settings can be displayed and changed. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

SOURCEFORMATTING - Source Map Formatting Option
MACROEXPANSIONS - Macro Expansions
FORMATTINGBIAS - Initial Formatting Bias
STORAGELOCATIONS - Storage Location Displays
DISPLACEMENTFIELDS - Displacement Field Displays

TEXTENCODING - Display text in EBCDIC or ASCII
DISPLAYWIDTHS - Storage and Register Display Widths
POINTERRESOLUTIONS - Resolve Pointers in Formatted Storage Displays
TRACEPOSITIONING - Trace Display Positioning
TRACEBASR - Trace BASR Into/Over Default
BREAKPOINTOPCODE - Breakpoint Opcode
DEADTRAPBYPASS - DEAD Trap Bypass Option for T BY
FINDFORMATTING - Display Option for the FIND Command

When you have finished changing parameters on the **10 FORMAT TRACE and FIND Settings** submenu, you can press:
- **F3** to **accept** your changes and exit the current menu.
- **F4** to **reject** your changes and exit back to the Home menu.
- **F5** to **accept** your changes and exit the **entire** Profile Menu System.

If you've made a mistake and accepted changes when you didn't really mean to, you can still revert to the prechange settings by issuing a **PROFILE READ name** command to reload the unchanged profile from disk. This is because all the changes you may have made are not actually hardened until you issue a **PROFILE SAVE** command.

But once you do issue a **PROFILE SAVE [name]** command, all profile changes are hardened and can be reloaded in future debugging sessions. See **HELP COMMANDS PROFILE SAVE** for more information.

Help PRofiles MEnu Formattracefind SOurceformatting

The "10 FORMAT TRACE and FIND Settings" submenu has a setting captioned **Source Map Formatting Option.**

This setting affects the formatted displays produced by the **FORMAT, WHERE, EWHERE** and **FIND** commands when the storage being displayed is mapped by a source listing map (such as an **ADATA** map or a **DWARF** data map). The choices are:

SOURCE - The displays produced by the commands are to incorporate all of the information that's available from the map: source statement images, statement labels, commentary, nearly everything.

Additionally, if Z/XDC detects a mismatch between the information in the map, and the object code in storage, it will display that piece of storage **twice:** Once showing information from the map, and again showing a disassembly of actual storage.

OBJECT - The displays are to show only disassembled storage but still labeled with statement label information taken from the map. All other information from the map is not used.

BOTH - The displays are to show both all available information from the map as well as the disassembly of actual storage. Thus, each location in storage is shown twice: Once showing information from the map, and

again showing the disassembly.

The default setting in all Factory Default profiles is **SOURCE.**

The equivalent Z/XDC command is **SET FORMAT SOURCE/OBJECT/BOTH.**

This setting only sets the **default** for the **FORMAT, WHERE, EWHERE** and **FIND** commands. These defaults can be overridden via operands on the commands themselves.

This setting has effect only when displaying areas of storage covered by **ADATA** maps and **DWARF** data maps. For storage that has not been mapped, and for storage that is covered only by **SYM data** maps, this setting has no effect.

Help PRofiles MEnu Formattracefind Macroexpansions

The "10 FORMAT TRACE and FIND Settings" submenu has a setting captioned **Macro Expansions.**

When:

- An area of storage is being displayed by the **FORMAT, WHERE, EWHERE** or **FIND** commands,
- And an **ADATA** map (not SYM data or DWARF data maps) has been assigned to that area,
- And the ADATA map is for **code,** not data,

Then this setting controls whether or not the display will include code generated from macro expansions.

The choices are:

SHOWMCODE - Code generated by macro expansions is **included** in the display.

HIDEMCODE - Macro generated code is **excluded** from the display.

CURRENTMCODE - Macro expansion code is displayed or suppressed according to whether a **PSW** (error level or retry level) points into the expansion. If one does, then the macro expansion is displayed. If one doesn't, then the macro expansion is suppressed.

The default setting in all Factory Default profiles is **CURRENTMCODE.**

The equivalent Z/XDC command is **SET FORMAT SHOWMCODE** or **HIDEMCODE** or **CURRENTMCODE.**

This setting only sets the **default** for the **FORMAT, WHERE, EWHERE** and **FIND** commands. These defaults can be overridden via operands on the commands themselves.

For more information, see **HELP MAPS ADATA MACROS.**

Help PRofiles MEnu Formattracefind FOrmattingbias

The "10 FORMAT TRACE and FIND Settings" submenu has a setting captioned **Initial Formatting Bias.** This controls the **FORMAT** (and friends) command's initial formatting bias.

When the **FORMAT, WHERE, EWHERE** and **SHOW** commands display storage, they generally prefer to start off attempting to disassemble that storage into machine instructions. But if the initial byte being displayed is not a valid opcode, then the display will automatically fall back to a hexadecimal-text display format.

However, that preference is only an **initial** preference. There are many influences (information from maps, from equates, from module boundaries, etc.) that these commands consider in an attempt to make a correct initial formatting choice.

Unfortunately, complete information often is not available; therefore, it simply is not possible for Z/XDC to always make the correct choice. So this setting can be used to change the default initial formatting preference. The choices are:

DATA - The **FORMAT, WHERE, EWHERE** and **SHOW** commands will always start off displaying storage in a hexadecimal-text format. But as the display progresses, if information is encountered that indicates that formatting should reattempt disassembly, then disassembly will be reattempted.

INSTRUCTION - The **FORMAT, WHERE, EWHERE** and **SHOW** commands will unconditionally begin their displays with an attempt to show disassembled machine instructions. All indications and influences to the contrary will be ignored **except** if the first byte to be disassembled is not a valid opcode.

NOBIAS - When making its initial formatting decision, the **FORMAT, WHERE, EWHERE** and **SHOW** commands will take into consideration all available influences, and make its own decision based on the result.

The default setting in all Factory Default profiles is **NOBIAS.**

The equivalent Z/XDC command is **SET FORMAT INSTRUCTION/DATA/NOBIAS.**

This setting only sets the **default** for the **FORMAT, WHERE, EWHERE** and **SHOW** commands. These defaults can be overridden via operands on the commands themselves. But typically, you would want to leave this default setting as **NOBIAS,** and only override it for individual commands on an as needed basis.

Help PRofiles MEnu Formattracefind SToragelocations

The "10 FORMAT TRACE and FIND Settings" submenu has a setting captioned **Storage Location Displays.**

This affects how storage locations are shown when the **FORMAT, SHOW, WHERE,**

EWHERE and **DISPLAY** commands are used to display storage.

The choices are:

ADDRESS - The storage locations of the displays are shown, running down the left side of the display, as 8-digit (4-byte) hexadecimal addresses. (The full 16-digit address for the start of the display is shown only in the display's header lines.)

OFFSET - The storage locations are shown as offsets from the start of some hopefully appropriate object within which the display falls. Such an object might be:
- A load module
- A csect
- A mapped control block
- An equate with a length attribute greater than **31**

Z/XDC does its best to make a good choice. **However,** if Z/XDC does make a choice that you don't like, you can change the offset base and range by creating an **equate,** located at your preferred base address and having a length attribute set to the area of storage in which you want the displayed offsets corrected. This will work as long as the equate's length is longer than X'20' bytes. For more information, see **HELP COMMANDS EQUATE.**

The default setting in all Factory Default profiles is **OFFSET.**

The equivalent Z/XDC command is **SET FORMAT OFFSETS/ADDRESSES.**

This setting only sets the **default** for the **FORMAT, WHERE, EWHERE, SHOW** and **DISPLAY** commands. These defaults can be overridden via operands on the commands themselves.

But typically, you would want to leave this default setting as **OFFSET,** and only override it for individual commands on an as needed basis. (In my own experience, the place where I most frequently use a local override for this setting is on the **SHOW** command.)

Help PRofiles MEnu Formattracefind DISPLACementfields

The "10 FORMAT TRACE and FIND Settings" submenu has a setting captioned **Displacement Field Displays.** This controls how displacement fields (a machine instruction's **ddd** fields) are shown when Z/XDC disassembles machine instructions from storage.

(The displays of machine instructions taken from **ADATA** maps are not affected.)

The choices are:

HEXADECIMAL - The displacement fields are shown in X-quoted hex. Example: L R14,X'132'(,R14)

DECIMAL - The displacement fields are shown in unframed decimal. Example: L R14,306(,R14)

The default setting in all Factory Default profiles is **HEXADECIMAL.**

The equivalent Z/XDC command is **SET FORMAT HEXADECIMAL/DECIMAL.**

Note that a machine instruction's hexadecimal object code is always shown at the left, so choosing **HEXADECIMAL** for this setting is, admittedly, redundant.

Nonetheless, it is my personal preference. Your preference might differ; in which case, you will want to change this setting.

Help PRofiles MEnu Formattracefind TExtencoding

The "10 FORMAT TRACE and FIND Settings" submenu has a setting captioned **Text Encoding for Raw Data Displays.** This controls whether text data is displayed as **ASCII** or **EBCDIC.**

Text Encoding for Raw Data Displays

When data is displayed by any command **(FORMAT, WHERE, EWHERE, SHOW, DISPLAY** and various **LIST** commands) in a hexadecimal-text format, this setting controls whether the text portion of the display shows an **EBCDIC** interpretation of storage, or an **ASCII** interpretation.

Notice: This setting also affects how Z/XDC translates text strings given on the **COPY, FIND, VERIFY** and **ZAP** commands. See **HELP COMMANDS SYNTAX STRINGDATA** for more information.

The default setting in all Factory Default profiles is **EBCDIC.**

The equivalent Z/XDC command is **SET FORMAT EBCDIC/ASCII.**

When storage is being interpreted as **EBCDIC,** the displayed text is framed by asterisks (*). When **ASCII** is being used, the text is framed by vertical bars (|).

Help PRofiles MEnu Formattracefind DISPLAYwidths

The "10 FORMAT TRACE and FIND Settings" submenu has a setting captioned **Storage and Register Display Widths.** This controls whether Z/XDC's displays are formatted for **wide** or **narrow** 3270s:

- A **narrow** 3270 is **80** columns wide.
- A **wide** 3270 is wider, up to **160** columns wide.

WIDE works best when your 3270 is set to **130** columns or wider.

When storage is displayed in a hexadecimal-text format, and when register sets are displayed, this setting controls how wide the displays should be. The choices are:

WIDE - Z/XDC's storage displays are up to **130** characters wide, with each line of the display showing up to **32** bytes (8 words) of storage. This setting is suitable for users who are using terminals with large display geometries. See **HELP FULLSCREEN TERMINALS GEOMETRIES** for more information.

NARROW - The storage displays are reduced to being **77** characters wide with each line of the display showing only **16** bytes (4 words) of storage. This setting is suitable for users who are using terminals with displays that are only **80** characters wide.

The default settings are as follows:

Factory Default Profile	**Setting**
AWIDE	WIDE
A80	NARROW
CWIDE	WIDE
C80	NARROW

The equivalent Z/XDC command is **SET FORMAT WIDE/NARROW.**

Help PRofiles MEnu Formattracefind Pointerresolutions

The "10 FORMAT TRACE and FIND Settings" submenu has a setting captioned **Resolve Pointers in Formatted Storage Displays.**

When Z/XDC produces a formatted display of programs, control blocks and data areas (via **FORMAT, [E]WHERE** and **SHOW),** those displays will often include 3-byte, 4-byte and 8-byte wide fields that contain addresses to other locations ("pointer fields"). This setting controls whether or not Z/XDC will resolve those pointers and tell you what they point to. The choices are:

- **POINTERS:** The pointed-to locations are resolved and identified.
- **NOPOINTERS:** The pointed-to locations not are resolved and identified.

The default setting in all Factory Default profiles is **POINTERS.**

The equivalent Z/XDC command is **SET FORMAT POINTERS** or **NOPOINTERS.**

Pointer Fields

Pointer fields are 3-byte, 4-byte and 8-byte wide bits of storage that contain the address of something else. The resolution of such pointer fields happens when:

- The field is contained in a csect, control block or data area.
- Said area is mapped by an Assembler csect of dsect map.
- The **OBJECT** operand is used, implied or forced on the **FORMAT, [E]WHERE** or **SHOW** command.
  - **OBJECT processing** causes storage to be formatted and displayed using Z/XDC's internal **disassembler.**
  - When the governing map is built from **SYM data,** OBJECT processing is always forced.
  - When the governing map is built from **ADATA,** OBJECT processing has to be requested (via OBJECT or BOTH operands) or implied (via a prior SET FORMAT OBJECT or BOTH command).
  - Pointer field resolutions do not occur when **SOURCE** formatting occurs.
- When **OBJECT processing** occurs, Z/XDC displays storage using its built-in disassembler. It is this disassembler that performs the pointer field resolutions. (The resolutions do not occur when storage is displayed with source image formatting.)

Performance Considerations

When very complex structures of **floating maps and equates** are present, the **POINTERS** setting may result in slower displays. In that case, you may wish to use **NOPOINTERS** instead. (For more information about floating maps and equates, see **HELP MAPS FLOATING.)**

Most of the storage display commands **(DISPLAY, FORMAT, SHOW, WHERE,** and **FIND)** accept operands that allow these settings to be temporarily overridden when the command is issued.

Help PRofiles MEnu Formattracefind TRACEPositioning

The "10 FORMAT TRACE and FIND Settings" submenu has a setting captioned **Trace Display Positioning.**

When stepping through code using the **TRACE** command, this controls whether the trace point runs down the display **(ROLL)** or the trace point remains at the top and the display scrolls up to it **(SCROLL).**

Trace Display Positioning

This setting controls how displays of storage are positioned during program execution tracing. The choices are:

SCROLL - During program execution tracing, this setting causes the program execution location to be always positioned at the top of the displays. Thus, at each step of the trace, the storage location being displayed always changes to start with the new program execution location.

ROLL - This setting attempts to maintain a more stable display. As you trace through program execution, the program execution location moves downward (usually) through the display, and the area of storage being displayed does not change until program execution moves beyond view. When that occurs, Z/XDC selects a new storage display position with the new program execution location positioned at the top of the display. This new display position will then be maintained until the execution location again moves out of view.

The default setting in all Factory Default profiles is **ROLL.**

The equivalent Z/XDC command is **SET TRACE ROLL/SCROLL.**

Help PRofiles MEnu Formattracefind TRACEBasr

The "10 FORMAT TRACE and FIND Settings" submenu has a setting captioned **Trace BASR Into/Over Default.** Its value may be set to **LOCAL** or **ALL.**

Which Traces Are and Are Not Affected by this Setting

This setting is effective:

- Only for **T, T I, T B** and **T BY** traces,
- And only when the branch target is located in **alien** storage. (See **HELP BREAKPOINTS TRACING ALIEN** for a definition of "alien".)

This setting does not affect **T BNC, T SB** and **T SA** traces. That's because these traces already try to stay within the local code, while the **T, T I, T B** and **T BY** traces have the potential of following execution into remote subroutines. For more information, see **HELP BREAKPOINTS TRACING CONDITIONAL.**

What the LOCAL/ALL Settings Do

When stepping through program execution with the **TRACE** command, when execution reaches a branching instruction that will branch, and when the location being jumped to is located in **alien** storage, then this setting controls how tracing is to proceed. The choices are:

ALL - Tracing should continue to the location being jumped to.

LOCAL - Tracing is disallowed from following execution into the alien location.

- Instead, if the branching instruction is a **linkage** instruction (BAKR, BAL, BALR, BAS, BASR, BASSM, BRAS, BRASL, JAS and JASL), then a subroutine is being called, and **recapture traps** are set at the presumed return points. (See **HELP COMMANDS TRACE BNC** for the details.) Then the subroutine is allowed to run without being followed. Tracing will resume when execution returns from the subroutine and execution is recaptured.

- On the other hand, if the branching instruction is anything else, then error message **DBC183E** is issued, and the trace is **aborted.**

- **Warning!** with this **LOCAL** setting, it is possible for Z/XDC to lose control of the target program's execution. For more information about this issue, see **HELP BREAKPOINTS TRACING CONDITIONAL FAILURES.**

- The default setting in all **Factory Default Profiles** is **LOCAL.**

- The equivalent Z/XDC command is **SET TRACE LOCAL/GLOBAL.**

Help PRofiles MEnu Formattracefind Breakpointopcode

- The "10 FORMAT TRACE and FIND Settings" submenu has a setting captioned **Breakpoint Opcode.** You can enter **ZERO** or **TRAP2** to to control whether Z/XDC:
  - Runs as an ESTAE-type or FRR recovery routine,
  - Or runs as a TRAP2 exit routine.

- When **ZERO** is set, **X'00'** opcodes are used for breakpoints. When user program execution reaches the breakpoint, a hardware program check occurs (PIC 0001), which becomes an 0C1 abend. If Z/XDC is properly set up as an **abend recovery routine** (ESTAE, ESTAEX, ESTAI, ARR or FRR), then it will receive control to handle the 0C1 as a breakpoint.

- When **TRAP2** is set, **TRAP2** opcodes (X'01FF') are used for breakpoints. When user program execution reaches the breakpoint, a hardware Trap Operation occurs. If Z/XDC is properly set up as a Trap Handler, then it will receive control to handle the Trap Operation as a breakpoint.

- For more information about breakpoint types, see **HELP BREAKPOINTS TYPES.**

- The default setting in all Factory Default profiles is **ZERO.**

- The equivalent Z/XDC command is **SET TRACE ZERO|TRAP2.**

Help PRofiles MEnu Formattracefind DEadtrapbypass

The "10 FORMAT TRACE and FIND Settings" submenu has a setting captioned **DEAD Trap Bypass Option for T BY.**

This option affects both the **TRACE BY** and **TRACE BNC** commands, but only when user program execution reaches a conditional branch (BC or JC instruction as generated by a **#DIE** macro) that may or may not cause execution to jump around (thus bypassing) a **DEAD trap.** The choices are:

STOP - If the conditional branch will, in fact, cause a branch (i.e. will cause execution to **bypass** the **DEAD trap),** execution will pause on that branch just like it would for any other successful branch instruction encountered by **TRACE BY** processing.

IGNORE - This setting prevents **TRACE BY** and **TRACE BNC** processing from ever pausing on a conditional branch that bypasses a **DEAD trap.**

The reasoning is, stopping on **DEAD traps** that will be bypassed is not a useful thing to do.

The default setting in all Factory Default profiles is **IGNORE.**

The equivalent Z/XDC command is **SET TRACE STOP/IGNORE.**

Note, if you use a lot of **#DIE** macros in your code (like I do), this option makes usage of the **TRACE BY** and **TRACE BNC** commands quite a bit more convenient. (If you don't use **#DIEs** at all, then this option will not matter to you.)

Help PRofiles MEnu Formattracefind FIndformatting

The "10 FORMAT TRACE and FIND Settings" submenu has a setting captioned **Display Option for the FIND Command.** This controls how the **FIND** command displays storage.

When the **FIND** command (which searches storage for a given string) finds what it is looking for, it wants to display what was found. It does this by internally issuing either a **FORMAT** command or a **DISPLAY** command to display the storage that it has found. This setting sets **FIND's** default initial display mode. (It can be overridden by an operand on the **FIND** command.) The choices are:

FORMAT - The **FIND** command displays its prize using a **FORMAT** command.

DISPLAY - The **FIND** command displays its prize using a **DISPLAY** command.

The default setting in all Factory Default profiles is **DISPLAY.**

There is no equivalent Z/XDC command for this setting. However, this setting can be overridden by using the **FIND** command's **FORMAT** or **DISPLAY** operands.

Help PRofiles MEnu Automapprintetc

The **11 AUTOMAP PRINT and Other Settings** submenu allows you to display and change a variety of settings including the **AUTOMAP** setting, **PRINT** command settings, and a few others.

Specifically, the following settings can be displayed and changed. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

- **AUTOMAP** - Controls whether or not module and csect maps are to be automatically loaded as needed
- **LPP** - The Lines Per Page setting for PRINT command output
- **SYSOUTCLASS** - The SYSOUT output class setting for PRINT command output
- **HOLDNOHOLD** - The HOLD/NOHOLD setting for PRINT command output
- **CANCELDETACHDEBUGGING** - Debugging option for ABENDs due to CANCEL and DETACH.
- **SETBANG** - Meaning of the ! Indirect Operator
- **CONVERSATIONTIMEOUT** - A Conversation Management control for when Z/XDC is debugging across multiple execution tasks
- **FLC** - Function leader Character for Built-in Functions

When you have finished changing parameters on the **AUTOMAP PRINT and Other Settings** submenu, you can press:
- **F3** to **accept** your changes and exit the current menu.
- **F4** to **reject** your changes and exit back to the Home menu.
- **F5** to **accept** your changes and exit the **entire** Profile Menu System.

If you've made a mistake and accepted changes when you didn't really mean to, you can still revert to the prechange settings by issuing a **PROFILE READ name** command to reload the unchanged profile from disk. This is because all the changes you may have made are not actually hardened until you issue a **PROFILE SAVE** command.

But once you do issue a **PROFILE SAVE [name]** command, all profile changes are hardened and can be reloaded in future debugging sessions. See **HELP COMMANDS PROFILE SAVE** for more information.

Help PRofiles MEnu Automapprintetc Automap

The "11 AUTOMAP PRINT and Other Settings" submenu has a setting captioned **AUTOMAP Programs Upon First Execution?**

You can enter **YES** or **NO** to turn auto-mapping on or off.

When **AUTOMAP** is turned on, then whenever Z/XDC receives control and user program execution is located within a new module and/or csect, Z/XDC will automatically load the **Binder** Map for the load module as well as the **ADATA, SYM data** or **DWARF data** map (if any) for the csect.

The default setting in all Factory Default profiles is **YES.**

The equivalent Z/XDC command is **SET AUTOMAP YES|NO.**

Help PRofiles MEnu Automapprintetc Lpp

The "11 AUTOMAP PRINT and Other Settings" submenu has a setting captioned **XDCPRINT Lines Per Page.**

This controls pagination in the **PRINT** command's output spool file. It sets the number of lines per printed page. The choices are:

nnn - A decimal number between 10 and 255. Page ejects and title line(s) are printed at the chosen interval.

0 - No pagination occurs. Page ejects and title lines are not inserted.

The default setting in all Factory Default profiles is **108** lines per page. This is a good number for printing onto 8.5x11 cutsheet paper.

The equivalent Z/XDC command is **SET PRINT LPP nnn.**

Help PRofiles MEnu Automapprintetc SYsoutclass

The "11 AUTOMAP PRINT and Other Settings" submenu has a setting captioned **XDCPRINT and SYSUDUMP output class**

This sets the **SYSOUT** class that the **PRINT** command should use for the output that it generates. Permitted values are, of course, A-Z 0-9 and * (the default output class for the job or TSO session in which the debugging session is occurring).

Warning! You should be sure to choose an output class that has not been set as a dummy class at your Data Center. (Ask your Systems Programmer for that information.)

The default setting in all Factory Default profiles is SYSOUT class **X.**

The equivalent Z/XDC command is **SET PRINT CLASS c.**

Help PRofiles MEnu Automapprintetc Holdnohold

The "11 AUTOMAP PRINT and Other Settings" submenu has a setting captioned **XDCPRINT Output Status.**

This controls whether or not the **PRINT** command's output spool file is to be set into a class hold state. The choices are:

HOLD - The output is forced into class hold state.
NOHOLD - The output is forced out of class hold state.
DEFAULT - The output either is or is not put into class hold state according to the SYSOUT class's default setting. (Ask your Systems Programmer for that information.)

The ddname used by the **PRINT** command is **//XDCPRINT.** See **HELP DDNAMES PRINT** for more information.

The default setting in all Factory Default profiles is **DEFAULT.**

The equivalent Z/XDC command is **SET PRINT HOLD YES|NO|DEFAULT.**

Help PRofiles MEnu Automapprintetc CAnceldetachdebugging

The "11 AUTOMAP PRINT and Other Settings" submenu has a setting captioned **Intercept/Debug CANCEL/DETACH ABENDs?**

You can enter **YES** or **NO** to control whether or not Z/XDC will allow you to debug abends arising from Operator **CANCEL** commands or task **DETACH** actions.

Normally, Z/XDC will intercept all ABENDs that it sees so that you have the opportunity to debug them. However, it generally is not useful to intercept ABENDs arising from CANCELs (x22 ABENDs) and DETACHs (x3E).

The default setting in all Factory Default profiles is **NO.**

The equivalent Z/XDC command is **SET CANDET DEBUG/IGNORE.**

Help PRofiles MEnu Automapprintetc SEtbang

The "11 AUTOMAP PRINT and Other Settings" submenu has a setting captioned **Meaning of ! Indirect Operator**

Historically, the exclamation point (!) has been used by Z/XDC in address expressions as an AMODE sensitive indirect operator. However, with the advent of 64-bit addressing, it became strongly desirable to change exclamation points to be 64-bit indirect operators. Thus, a dilemma arose: Existing users would have reason to want the old meaning, while new users would have reasons to expect the new meaning. So to resolve this problem, I implemented this setting to let individual users decide for themselves. The choices are:

AMODE - This causes exclamation points to represent AMODE sensitive indirect addressing.

64BIT - This causes exclamation points to represent 64-bit indirect addressing.

Note when **64BIT** is in effect, AMODE-sensitive indirect addressing can still be achieved via the **INDIRECT()** Built-in Function. See **HELP FUNCTIONS INDIRECT** for more information.

The default setting in all Factory Default profiles is **64BIT.**

The equivalent Z/XDC command is **SET BANG AMODE/64BIT.**

Help PRofiles MEnu Automapprintetc COnversationtimeout

The "11 AUTOMAP PRINT and Other Settings" submenu has a setting captioned **Multiconversation Timeout Control.**

This controls a delay that Z/XDC imposes between the time it releases control **(GO** or **TRACE** commands) back to the user program running in one task before it will accept a debugging request from another task.

You may enter a value ranging from 1 to 3600 seconds. The Factory Default in all profiles is **5** seconds.

Multiconversation Timeout Control

When debugging a multitasking program, it is quite possible for Z/XDC to receive control in more than one task at a time; however, there is only one display terminal connected to the debugging session, so only one task can be allowed to talk to the the terminal at a time.

During such a conversational thread, when **GO** or **TRACE** commands are used, Z/XDC will temporarily relinquish control back to the user program, at which time it would be possible for Z/XDC, waiting in another task, to grab control of the conversation. Such abrupt task switching is unbelievably undesirable.

So whenever Z/XDC releases control from one task (back to the user program), a timer is started to force other pending Z/XDC tasks to delay trying to grab control of the conversation. If Z/XDC regains control in the same task before the timer expires, then fine. The debugging conversation continues in that task.

However, if the time interval expires first, then a Z/XDC waiting in another task will grab control, and debugging will process under that new task.

This option controls the length of the delay period.

The equivalent Z/XDC command is **SET TIMEOUT nnn.**

Note: sometimes it may legitimately occur that the task you are trying to debug

takes too long to return control to Z/XDC. Then another task will grab control of the terminal. If this is undesirable, you can use the **WAIT** command to force the impatient task back into a wait state.

Later, when you finally want to turn your attention back to that impatient task, you can use the **THAW** command to take it out of the imposed wait state.

Help PRofiles MEnu Automapprintetc Flc

The "11 AUTOMAP PRINT and Other Settings" submenu has a setting captioned **Built-in Function Leader Character.**

This controls the special character used to introduce built-in functions in address expressions.

A syntax requirement of address expressions is that the names of built-in functions may (depending upon context) have to be introduced by a special character so as to delimit the name from other characters preceding the name in the expression. This character is called a **Function Leader Character (FLC).** This setting allows you to choose (within strict limits) what the **FLC** is to be. The choices are:

¢ - Cent sign
` - Back accent
~ - Tilde

The default setting in all Factory Default profiles is ~ (tilde).

The equivalent Z/XDC command is **SET FLC.**

Help PRofiles MEnu REXx

The **12 REXX settings** submenu allows you to display and change a variety of settings related to the Z/XDC REXX interface.

When you have finished changing parameters on the **12 REXX settings** submenu, you can press:

- **F3** to **accept** your changes and exit the current menu.
- **F4** to **reject** your changes and exit back to the Home menu.
- **F5** to **accept** your changes and exit the **entire** Profile Menu System.

If you've made a mistake and accepted changes when you didn't really mean to, you can still revert to the prechange settings by issuing a **PROFILE READ name** command to reload the unchanged profile from disk. This is because all the changes you may have made are not actually hardened until you issue a **PROFILE SAVE** command.

But once you do issue a **PROFILE SAVE [name]** command, all profile changes are hardened and can be reloaded in future debugging sessions. See **HELP COMMANDS PROFILE**

SAVE for more information.

Help PRofiles MEnu CXdcsettings

The **c/XDC Settings** submenu allows you to display and change a variety of settings relating to **c/XDC** support. These include settings that can also be set by various of the **SET** commands. Examples:

- **SET CXDC**
- **SET STEP**
- **SET VSETTINGS**
- **SET VDISPLAY**

When you have finished changing parameters on this menu, you can press:
- **F3** to **accept** your changes and exit the current menu.
- **F4** to **reject** your changes and exit back to the Home menu.
- **F5** to **accept** your changes and exit the **entire** Profile Menu System.

If you've made a mistake and accepted changes when you didn't really mean to, you can still revert to the prechange settings by issuing a **PROFILE READ name** command to reload the unchanged profile from disk. This is because all the changes you may have made are not actually hardened until you issue a **PROFILE SAVE** command.

But once you do issue a **PROFILE SAVE [name]** command, all profile changes are hardened and can be reloaded in future debugging sessions. See **HELP COMMANDS PROFILE SAVE** for more information.

Help PRofiles MEnu Uss

The **14 USS** menu allows you to display and change a variety of settings related to the Z/XDC USS interface.

When you have finished changing parameters on the **USS** menu, you can press:
- **F3** to **accept** your changes and exit the menu.
- **F4** to **reject** your changes and exit the menu.
- **F5** to **accept** your changes and exit the **entire** Profile Menu System.

If you've made a mistake and accepted changes when you didn't really mean to, you can still revert to the prechange settings by issuing a **PROFILE READ name** command to reload the unchanged profile from disk. This is because all the changes you may have made are not actually hardened until you issue a PROFILE SAVE command.

But once you do issue a **PROFILE SAVE [name]** command, then all profile changes are hardened and can be reloaded in future debugging sessions. See HELP COMMANDS PROFILE SAVE for more information.

Help PRofiles MApliblists

In addition to the items described under the HELP PROFILE MENU topic, Z/XDC can also save in the session profile one or more named lists of MAPLIB libraries.

A MAPLIB library is a sequential or partitioned dataset that contains information (usually ADATA) for use by the MAP and DMAP commands. A MAPLIB list is a list of one or more such datasets. See HELP MAPS for more information.

One or more MAPLIB lists can be saved in the session profile. In order to keep the lists distinguished from each other, a MAPLIB list needs to be assigned a name when it is saved.

- The **SET MAPLIBS** command can be used to:
  - Build an active MAPLIB list.
  - Save an active MAPLIB list into the session profile. (The PROFILE SAVE command will have to be used to make the save action permanent.)
  - Load a MAPLIB list from the session profile and activate it.
  - Purge the active MAPLIB list.

  See HELP COMMANDS SET MAPLIBS for more information.

- The **LIST MAPLIBS** command can be used to:
  - Display the active MAPLIB list.
  - Display a summary list of all saved MAPLIB lists.
  - Display the detailed description of any one or more saved MAPLIB lists.

- The **DELETE MAPLIBS** command can be used to:
  - Delete any one or more or all datasets from the active MAPLIB list.
  - Delete any one or more or all saved MAPLIB lists.

When a debugging session starts, if the initial session profile that is used by Z/XDC contains a MAPLIB list named "DEFAULT" then Z/XDC will automatically load that list and activate it. To see what has been loaded, use the **LIST MAPLIBS ALL** command shortly after starting a debugging session.

Note, Z/XDC’s factory default profiles do not provide any MAPLIB lists, default or otherwise. So in order to create a default MAPLIB list, you will have to use one or more SET MAPLIBS command to built a list, followed by **both** of the following commands:

SET MAPLIBS SAVE: To save the active MAPLIB list (as the default) into Z/XDC’s in-storage copy of the active profile.

PROFILE SAVE: To cause the current profile data to be written to disk for future use as your personal session profile. **WARNING!** Failure to issue the PROFILE SAVE command will cause any MAPLIB lists that you may have tried to save to be lost at the end of the debugging session.

Note: even though the MAPLIB lists are saved into the session profile, they cannot be displayed or manipulated by the Profile Menuing System. (See HELP PROFILE MENU.) They can be displayed and manipulated only by the SET, LIST, and DELETE commands discussed above.

However, the PROFILE RESET, READ, and SAVE commands do have the following effects:

PROFILE RESET name
This command causes Z/XDC to load one of its factory default profiles. Z/XDC does not have any factory default MAPLIB lists. Therefore the PROFILE RESET command causes any MAPLIB lists that you may have saved into your session profile to be unavailable. The command does not, however, have any effect on the active MAPLIB list.

PROFILE READ
This command causes Z/XDC to (re)load either the default session profile or a named session profile. Consequently, this command can change the set of saved MAPLIB lists available to the SET MAPLIBS READ command. This command does not, however, have any effect on the active MAPLIB list.

After issuing a PROFILE READ command, you can use the **LIST MAPLIBS ALL** command to see what lists have become available.

PROFILE SAVE
This command causes Z/XDC to write its current profile data to disk for use in future debugging sessions. It you wish to define and save one or more MAPLIB lists, then just issuing a SET MAPLIBS SAVE command is not enough.
You **MUST(!)** also issue a PROFILE SAVE command in order to make your lists permanently available. Failure to do so will cause your lists to be lost at the end of the current debugging session.

Note, the PROFILE SAVE command does not save the active MAPLIB list. It saves only the named lists that have been created by the SET MAPLIBS SAVE command.

Help PRofiles Readsave

Z/XDC does **not** automatically resave the profile every time you change a profiled setting. Thus, unless you explicitly issue a "PROFILE SAVE" command, all settings will be restored to your prior defaults the next time you issue a "PROFILE READ" command or start a new debugging session.

If you do wish to permanently change any of your profiled settings, be sure first to do a "PROFILE READ" so that you clear away any temporary settings that you may have forgotten about. Then make the changes that you want permanently profiled, then do a "PROFILE SAVE" to save those changes.

IF YOU CHANGE YOUR MIND: If you have saved a new default profile and later change your mind and decide that you don’t like it, there are several things you can do about it:

- You can type "PROFILE" to enter the "Profile Menuing System" and change whatever settings you want. When you END the Menuing System, be sure to type "PROFILE SAVE" to save your changes.

- You can restore to your Data Center’s "system-wide" default profile in either of two ways:
  - You can load the system’s default profile and resave it as your own personal profile with the following two commands:

        PROFILE READ XDC TFS

PROFILE SAVE

- Or you can use ISPF's "Library Utility" (option 3.1) to access your profile library and delete the member named **XDCDPV31.** (See HELP FULLSCREEN PROFILE DEFAULTPROFILE for an explanation of how default profiles are named in the profile library.)

- If you want to load the profile from an XDC clone of the current or a prior release, then specify the clone's name **(xxx)** as the PROFILE READ command's second operand:

  PROFILE READ XDC xxx

  (For more information about XDC clones, see HELP XDCCLONES.)

- If you want to restore one of Z/XDC's **factory default** profiles and save it as your personal profile, you can do so with the following two commands:

  PROFILE RESET [name]
  PROFILE SAVE

The factory default settings are hard-coded within Z/XDC and are the same for all customers. They are as described in several topics following HELP PROFILES MENU.

The **system-wide** default profiles are profiles for use by everyone at your Data Center. These profiles are distributed with settings that are identical to the Factory Default profiles; however, their settings may have been customized (by the Systems Programmer who installed Z/XDC) so as to be more suitable to your Data Center's local needs.

The **LIST PROFILES CURRENT** command shows the status of your active session profile.

The **contents** of your active profile can be displayed by typing the "PROFILE" command without operands. Doing so drops you into a Profile Menuing System from which you can easily display and change all profiled settings.

Exception: Saved MAPLIB lists cannot be displayed or changed from the Profile Menuing System. See HELP PROFILE MAPLIBLISTS for more information.

Profiled options (including MAPLIB lists) can also be displayed and changed individually via a variety of LIST, SET, and other commands. However, using the Profile Menuing System described above often is easier.

For related information, select the following topics. These topics will be displayed only if selected.

HELP COMMANDS SET PROFILE
HELP COMMANDS LIST PROFILES
HELP COMMANDS PROFILE

Help PRofiles Namedprofiles

Whenever you start a debugging session, Z/XDC provides you with a default profile nicknamed **XDC.** This default profile can be loaded from any of several sources. For related information about that, see **HELP PROFILE DEFAULTPROFILE.**

Z/XDC actually permits you to create and save any number of different session profiles. To do so, simply define the settings that you want and then type **PROFILE**

SAVE name where **name** is a five character name of your choice (subject, of course, to normal syntax requirements). Since you can choose any name, you can save any number of profiles.

To read a named profile, just type **PROFILE READ name.** Later, if you want to reload your personal default profile, you can type either **PROFILE READ XDC** or just **PROFILE READ.**

Window Layouts

So why would you want or need lots and lots of profiles? Most settings are not ones that you would typically want to change often.

However, there is one setting in particular that you might well want to change frequently and that makes multiple profiles a powerful debugging tool. That setting is the **Layout** of Watch Windows at your terminal.

One of the items saved in session profiles is the locations **and contents** of Watch Window command lines. This means that if you customize your screen's window layout for a particular debugging situation, then you can save that layout in a named profile and restore it instantly simply by typing **PROFILE READ name.** It also means that if you move Watch Windows around for some temporary purpose, you can restore your normal layout just by typing **PROFILE READ.**

Watch Windows can be created and destroyed very easily with the **SET WINDOW CREATE** command and the **SET WINDOW DELETE** command (factory default PF keys shift-F1 and F1, respectively).

- **To create a Watch Window,** just move the cursor to where you want its command line to be, then press shift-F1. A new command line will appear. Then type onto that command line any DISPLAY, FORMAT, LIST, SHOW, and/or WHERE commands you want.

- **To remove a Watch Window,** move the cursor to anywhere within it, then press F1. The command line will disappear.

- **To save the layout that you have created,** home the cursor to the screen's top command line, and type "PROFILE SAVE name". The layout, along with all other current settings, will be saved in the profile that you name, and it can later be restored just by typing "PROFILE READ name".

Watch Windows are a perfect way to keep track of particular registers, fields and data areas in storage, and anything else that Z/XDC can display. A particular layout of Watch Windows, once created and saved, can be reloaded either manually (by typing the "PROFILE READ name" command) or automatically at breakpoints by assigning the "PROFILE READ name" command as a breakpoint's associated **automatic command.** (See **HELP BREAKPOINTS AUTOCMDS.)**

Subtopics Index

To see a specific example, type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

EXAMPLE - Example of making use of multiple named profiles.

Help PRofiles Namedprofiles Example

The following is an description of a time when I used multiple named profiles in a debugging situation where each named profile contained a different layout of Watch Windows.

- I needed to debug a program containing a very large loop.
- I needed to set breakpoints at six places around the loop,
- At each breakpoint I needed to display information that was specific to that particular part of the loop.

To do this, I did the following:

- I used PF keys F1 and shift-F1 (PF key 13) to create six different window layouts (one at a time), each with the appropriate **DISPLAY, FORMAT, LIST, SHOW,** and **WHERE** commands to show the specific information that I needed to see at that point.

- As I created each of these layouts, I used the **PROFILE SAVE name** command to save each of them under their own names (e.g. **PROFILE SAVE LPTOP, PROFILE SAVE READX,** etc.).

- I then used the **AT** command to define the six breakpoints that I needed,

- On each of the **AT** commands, I associated the **PROFILE READ name** command that would load the particular layout that I wanted to see at that point of the loop.

  Example:
  - **AT .TOPOFLP 'PROF READ LPTOP;WHERE'.**
  - I included the **WHERE** command to cause the breakpoint location itself to be displayed in the screen's Primary Window.

- Finally, I used a **GO** command to let my program resume execution and enter its loop.

When my program reached the first breakpoint, Z/XDC regained control, and the desired display layout was automagically loaded and displayed.

To advance around the loop, stopping at each breakpoint along the way, all I had to do was to keep on issuing **GO** commands.

Help PRofiles Profnames

When you use the name of a profile on a **PROFILE READ** or a **PROFILE SAVE** command, the given name must conform to the following syntax rules:

- It must be from 1 to 5 characters long.
- It must consist of alpha, numeric, and national characters ($ # @).

- The first character may not be numeric.

When a named profile is written to a profile library member, Z/XDC constructs the member name as follows:

- The member name's first three characters match Z/XDC's name. For example, if you are using an Z/XDC that was renamed during installation to **V31,** then the names of all profile library members written by that Z/XDC will start with the characters **V31.**
- The member name's remaining 5 characters will match the name given on the **PROFILE READ** or **PROFILE SAVE** command.

Example: If you are running a Z/XDC that has **not** been renamed (as I'm sure most of you are), and if you type **PROFILE SAVE WAX58,** then Z/XDC will create in your profile library a member named **XDCWAX58.**

When a named profile is read from a profile library, Z/XDC makes two attempts to find the profile:

- First, Z/XDC searches the profile libraries for a member whose name matches the given name prefixed by Z/XDC's current name. Example: If you are using a Z/XDC that has been renamed to **V31** and you type **PROFILE READ WORK1,** then Z/XDC will search the profile libraries for a member named **V31WORK1.**
- If the first search fails, then Z/XDC retries by prefixing the given name with **TFS** and searching again. In the above example, if Z/XDC cannot find **V31WORK1,** then it will try again searching for **TFSWORK1.**

Z/XDC reports "profile not found" only if both searches fail.

For a discussion of the libraries searched by **PROFILE READ/SAVE** processing, see HELP PROFILE DDNAMES.

If you should need to read a profile whose member name's first three characters do **not** match Z/XDC's current name, you can simply give those three characters as an additional operand on the **PROFILE READ** command. For example, if you need to load a profile whose library member name is **XYZWAX58,** then typing **PROFILE READ WAX58 XYZ** does the trick.

XDC, when used as a profile name, is a special case. The **PROFILE** commands interpret **XDC** to mean "the default profile"; accordingly, Z/XDC searches for a special set of library members whenever the profile name **XDC** is specified or implied.

Note that the **PROFILE READ** command and the **PROFILE READ XDC** command are identical in what they do. For more information, proceed to the following topic (HELP PROFILES DEFAULTPROFILE) by typing HELP ∗NEXT (F11).

Help PRofiles DEfaultprofile

Whenever you start a Z/XDC debugging session, a default session profile is automatically loaded from... somewhere. Just where that somewhere is depends on a variety of things.

When dump/XDC is Being Used

When Z/XDC is running under the control of **dump/XDC** (i.e. when Z/XDC is being used to examine a dump), dump/XDC may (or may not) provide initial session profile settings for Z/XDC to use:

- When dump/XDC is started, it is possible for the user to tell dump/XDC what initial session profile it is to tell Z/XDC to use. This is done via a **PROFILE(name)** parameter on the **%DXDC** command used to start dump/XDC. See **HELP DUMPS STARTINGDUMPXDC DXDCPROC** for the details.

- Absent that, if dump/XDC is being used to resume the examination of a previously examined dump, then dump/XDC will provide the profile settings to Z/XDC based on their values in use the last time that dump was examined. (dump/XDC saves profile settings on a per-dump basis, so settings in use for one dump do not affect those in use for another.) See **HELP DUMPS MISCELLANEOUS PROFILES** for more information.

- Absent that (for example, when dump/XDC is being used to examine a new dump), then prior profile settings do not exist, so dump/XDC allows Z/XDC to start up without prior settings. In this case, Z/XDC will engage its own processing for finding an initial session profile. (See below.)

When dump/XDC is Not Being Used (or Just Doesn't Care)

When Z/XDC is running normally (for example when Z/XDC is not under the control of dump/XDC), it has to choose, find and load its **default** session profile from a profile library (ddnames XDCPROF or ISPPROF) or a table library (XDCTLIB or ISPTLIB) or Factory Defaults (hardcoded within Z/XDC's source code).

The name of the library **member** that Z/XDC loads and uses for its default profile changes with each new release of Z/XDC; however, from **within** Z/XDC the default profile can always be referenced simply by the name **XDC.** This is:

- Regardless of what **release** of Z/XDC you are running,
- And regardless of whether you are running XDC itself or a **clone** of XDC renamed to something else.

(This special name conversion does **not** occur for any given profile name other than the name **XDC.)**

Accordingly, whenever you need to reload your default profile implicitly, you can type **either** of the following commands:

```
PROFILE READ
PROFILE READ XDC
```

For Z/XDC's current release (v3.1), the name of the profile library member that Z/XDC uses for its defaults is **xxxDPV31** (where **xxx** matches the current Z/XDC's clone name). All of the following commands will use this member either implicitly or explicitly:

```
PROFILE READ
PROFILE READ XDC
PROFILE READ DPV31
```

If a default profile library member does not exist, then what Z/XDC does next depends upon which of the above three commands was used.

If **PROFILE READ DPV31** was used, Z/XDC searches the profile libraries for the following members in the following order:

xxxDPV31 - (where **xxx** is Z/XDC's current name)
TFSDPV31 - (usually created at Z/XDC installation time and placed into an ISPF table library)

On the other hand, if either **PROFILE READ** or **PROFILE READ XDC** was used (and xxxDPV31 does not exist), then Z/XDC automatically looks for a default profile member for an **earlier release** of Z/XDC. Specifically, Z/XDC searches profile libraries (such as ISPPROF) and table libraries (such as ISPTLIB) for the following profile library members in the following order:

xxxDPV31 (v3.1's default profile)
xxxDPZ22 (z2.2's default profile)
xxxDPZ21 (z2.1's default profile)
xxxXDC1D (z1.13's default profile)
xxxXDC1C (z1.12's default profile)
xxxXDC1B (z1.11's default profile)
xxxXDC1A (z1.10's default profile)
xxxXDC19 (z1.9's default profile)
xxxXDC18 (z1.8's default profile)
xxxXDC17 (z1.7's default profile)
xxxXDC16 (z1.6's default profile)
xxxXDC13 (z1.3's default profile)
xxxXDC12 (z1.2's default profile)
xxxXSE21 (S2.1's default profile)
xxxXSE20 (S2.0's default profile)
xxxXSE11 (S1.1's default profile)
xxxXSE10 (S1.0's default profile)
xxxXDC34 (X3.4's default profile)
xxxXDC33 (X3.3's default profile)
xxxXDC32 (X3.2's default profile)
xxxXDC31 (X3.1's default profile)
xxxXDC30 (X3.0's default profile)
xxxXDC22 (X2.2's default profile)
xxxATT21 (X2.1's default profile)
xxxATTR (X2.0's default profile)

If **none** of these members is found then Z/XDC re-searches the libraries for the following system-wide default profile members in the following order:

TFSDPV31 (v3.1's customer default)
TFSDPZ22 (z2.2's customer default)
TFSDPZ21 (z2.1's customer default)
TFSXDC1D (z1.13's customer default)
TFSXDC1C (z1.12's customer default)
TFSXDC1B (z1.11's customer default)
TFSXDC1A (z1.10's customer default)
TFSXDC19 (z1.9's customer default)
TFSXDC18 (z1.8's customer default)
TFSXDC17 (z1.7's customer default)
TFSXDC16 (z1.6's customer default)
TFSXDC13 (z1.3's customer default)
TFSXDC12 (z1.2's customer default)
TFSXSE21 (S2.1's customer default)
TFSXSE20 (S2.0's customer default)
TFSXSE11 (S1.1's customer default)
TFSXSE10 (S1.0's customer default)
TFSXDC34 (X3.4's customer default)

```
TFSXDC33   (X3.3's customer default)
TFSXDC32   (X3.2's customer default)
TFSXDC31   (X3.1's customer default)
TFSXDC30   (X3.0's customer default)
TFSXDC22   (X2.2's customer default)
TFSATT21   (X2.1's customer default)
TFSATTR    (X2.0's customer default)
```

When No Profile Can be Found

If after this exhaustive search a default profile still cannot be found, then Z/XDC finally falls back to using one of its hard-coded **Factory Default** profiles. See **HELP PROFILES FACTORYDEFAULTS** for details.

If you wish to load a Factory Default profile explicitly, then you can do so by issuing the the **PROFILE RESET [name]** command.

Profiles from Older Releases

Whenever you issue a **PROFILE READ XDC** command, if the default profile for Z/XDC's current release does not exist, a potentially lengthy search is made for an older release's default profile. If such is found, it is loaded and converted, but it is **not** automatically saved! That doesn't happen until you explicitly issue a **PROFILE SAVE XDC** command, at which time, a **xxxDPV31** member will be created in your profile library. For more information, see **HELP PROFILES OLDPROFILES.**

Help PRofiles Factorydefaults

Factory Default profiles are profiles whose various settings are hardcoded into templates that have been assembled into Z/XDC's source code. There are **four** Factory Default profiles. They are:

A80 - This is suitable for Assembler programmers using 80-column wide terminals.

AWIDE - This is suitable for Assembler programmers using terminals that are around **136** columns wide or wider.

C80 - This is suitable for C programmers using 80-column wide terminals.

CWIDE - This is suitable for C programmers using terminals that are around **136** columns wide or wider.

Subtopics Index

For details about each of these profiles, Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

For details about the **PROFILE RESET** command, see HELP COMMANDS PROFILE RESET.

For details about the **PROFILE READ** command, see HELP COMMANDS PROFILE READ.

Help PRofiles Factorydefaults A80

This Factory Default profile is named **A80.** It is suitable for use by Assembler programmers using 80-column wide terminals. It is packaged as a hard-coded template built into Z/XDC's Assembler source code. It can be loaded by the **PROFILE RESET A80** command. See **HELP COMMANDS PROFILE RESET** for details.

Of the several dozen settings found in each profile, only a few differ from one Factory Default profile to the next. For those settings that are the same across each profile, see the several topics that follow **HELP PROFILES MENU.** For those settings that differ, see below.

Display Width

This **A80** profile issues **SET FORMAT NARROW.** This causes Z/XDC to format most of its displays to fit within 80 columns.

Display Layout

All Factory Default profiles divide the terminal's display area into two windows. The top window is the **Primary Window,** while the second window is a **Watch Window.** (For more information, see **HELP FULLSCREEN WINDOWS.)**

The Factory Default profiles differ between wide and narrow displays in the definition of the Watch Window:

- For the **C80** and **A80** profiles,
  - the Watch Window command line contains: **L PSW;L EPSW;L REGS.**
  - The window is 7 rows high.

- For the **CWIDE** and **AWIDE** profiles,
  - the Watch Window command line contains: **L PSWE;L EPSWE;L BEA;;L RWREGS;;L AREGS**
  - The window is 11 rows high.

Latent Commands String

The Factory Default profiles all set the same Latent Commands String: **L PSWE;L BEA;L RWREGS.** For more information about Latent Command Strings, see **HELP FULLSCREEN LOGGING LATENTCOMMANDS.**

Tracing PF keys

For all Factory Default profiles, most PF keys are defined identically. (See **HELP FULLSCREEN PFKEYS DEFAULTKEYS** for details.) Only the following keys differ:

9th - (F9) **T**
This is one of three PF keys that assist in the tracing of Assembler program execution. This allows you to step through the execution of your program exactly one machine instruction at a time.

10th - (F10) **T BY**
This is one of three PF keys that assist in the tracing of Assembler program execution. This allows you to step through the execution of your program from one successful branch-type instruction to the next.

11th - (F11) **T NXSEQ() NXSEQ(2) NXSEQ(3) NXSEQ(4) NXSEQ(5) NXSEQ(6);GOT**
This is one of three PF keys that assist in the tracing of Assembler program execution. This PF key can be used to bypass the tracing of subroutines. If, during **T** or **T BY** tracing, execution reaches a **branch and link type** instruction, (see **BAL-type** instructions in **HELP BREAKPOINTS TRACING BRANCHES),** you have two choices of what to do next:

(a) - You may want to continue tracing execution into the subroutine, in which case, just continue to use the **T** or **T BY** (and other tracing commands) to proceed normally.

(b) - You may want to let the subroutine execute without being traced, but you want to recapture control of execution when the subroutine terminates. This can be accomplished by using this PF key.

Loops: This PF key also provides one way in which to bypass the tracing of repeated iterations of a loop. If you have reached the bottom instruction of a loop, and if you **know** that the loop eventually will exit to the next instruction following its bottom instruction, then use this PF key. **Warning:** if the loop should unexpectedly exit by branching to some other instruction, then you will lose control of the trace unless you take the precaution of first placing traps at all possible exit points.

This PF key definition is as complex as it is because it is designed to be usable for bypassing even subroutines that make vectored returns to their callers. Generally, such subroutines will return into a list of instructions immediately following the linking instruction.

However, this PF key can still be used for subroutines that make only normal returns. This is because the breakpoints set by this PF key are transient and will all be automatically removed by Z/XDC when the subroutine returns to the instruction just past its linking instruction.

Creating a Local Version of A80

If a Data Center wishes to make changes to this profile, a SysProg can load it (via the **PROFILE RESET A80** command) and make the desired changes and then save it via a **PROFILE SAVE A80 TFS** command. This creates a member named **TFSA80** in the SysProg's personal profile library.

To make this profile available System wide, the SysProg must then **move** it (not copy it) over to a suitable **//ISPTLIB** library. (The move must be done outside of Z/XDC.)

Then to load the modified profile, a programmer must use a **PROFILE READ A80** command (not PROFILE RESET A80).

Now in order to access the **TLIB** version of the A80 profile, the program that you are debugging must have an **//ISPTLIB** allocation to the appropriate ISPF Table Library. For debugging in TSO, that's usually automatic. For debugging batch jobs, however, you will need to add an **//ISPTLIB** DD card to your job's JCL. (This is true even when connecting to the debugging session from a TSO session!) For more information about ISPF-style **TLIBs,** see **HELP PROFILES TLIBPROFILES.**

Help PRofiles Factorydefaults AWide

This Factory Default profile is named **AWIDE.** It is suitable for use by Assembler programmers using terminals defined as having **136** columns or more. It is packaged as a hard-coded template built into Z/XDC's Assembler source code. It can be loaded by the **PROFILE RESET AWIDE** command. See **HELP COMMANDS PROFILE RESET** for details.

Of the several dozen settings found in each profile, only a few differ from one Factory Default profile to the next. For those settings that are the same across each profile, see the several topics that follow **HELP PROFILES MENU** For those settings that differ, see below.

Display Width

This **AWIDE** profile sets **SET FORMAT WIDE.** This causes Z/XDC to format most of its displays to fit within 136 columns.

Display Layout

All Factory Default profiles divide the terminal's display area into two windows. The top window is the **Primary Window,** while the second window is a **Watch Window.** (For more information, see **HELP FULLSCREEN WINDOWS.)**

The Factory Default profiles differ between wide and narrow displays in the definition of the Watch Window:

- For the **C80** and **A80** profiles,
  - the Watch Window command line contains: **L PSW;L EPSW;L REGS.**
  - The window is 7 rows high.

- For the **CWIDE** and **AWIDE** profiles,
  - the Watch Window command line contains: **L PSWE;L EPSWE;L BEA;;L RWREGS;;L AREGS**
  - The window is 11 rows high.

Latent Commands String

The Factory Default profiles all set the same Latent Commands String: **L PSWE;L BEA;L RWREGS.** For more information about Latent Command Strings, see **HELP FULLSCREEN LOGGING LATENTCOMMANDS.**

Tracing PF keys

For all Factory Default profiles, most PF keys are defined identically. (See **HELP FULLSCREEN PFKEYS DEFAULTKEYS** for details.) Only the following keys differ:

9th - (F9) **T**
This is one of three PF keys that assist in the tracing of Assembler program execution. This allows you to step through the execution of your program exactly one machine instruction at a time.

10th - (F10) **T BY**
This is one of three PF keys that assist in the tracing of Assembler program execution. This allows you to step through the execution of your program from one successful branch-type instruction to the next.

11th - (F11) **T NXSEQ() NXSEQ(2) NXSEQ(3) NXSEQ(4) NXSEQ(5) NXSEQ(6);GOT**
This is one of three PF keys that assist in the tracing of Assembler program execution. This PF key can be used to bypass the tracing of subroutines. If, during **T** or **T BY** tracing, execution reaches a **branch and link type** instruction, (see **BAL-type** instructions in **HELP BREAKPOINTS TRACING BRANCHES),** you have two choices of what to do next:

(a) - You may want to continue tracing execution into the subroutine, in which case, just continue to use the **T** or **T BY** (and other tracing commands) to proceed normally.

(b) - You may want to let the subroutine execute without being traced, but you want to recapture control of execution when the subroutine terminates. This can be accomplished by using this PF key.

Loops: This PF key also provides one way in which to bypass the tracing of repeated iterations of a loop. If you have reached the bottom instruction of a loop, and if you **know** that the loop eventually will exit to the next instruction following its bottom instruction, then use this PF key. **Warning:** if the loop should unexpectedly exit by branching to some other instruction, then you will lose control of the trace unless you take the precaution of first placing traps at all possible exit points.

This PF key definition is as complex as it is because it is designed to be usable for bypassing even subroutines that make vectored returns to their callers. Generally, such subroutines will return into a list of instructions immediately following the linking instruction.

However, this PF key can still be used for subroutines that make only normal returns. This is because the breakpoints set by this PF key are transient and will all be automatically removed by Z/XDC when the subroutine returns to the

instruction just past its linking instruction.

Creating a Local Version of AWIDE

If a Data Center wishes to make changes to this profile, a SysProg can load it (via the **PROFILE RESET AWIDE** command) and make the desired changes and then save it via a **PROFILE SAVE AWIDE TFS** command. This creates a member named **TFSAWIDE** in the SysProg's personal profile library.

To make this profile available System wide, the SysProg must then **move** it (not copy it) over to a suitable **//ISPTLIB** library. (The move must be done outside of Z/XDC.)

Then to load the modified profile, a programmer must use a **PROFILE READ AWIDE** command (not PROFILE RESET AWIDE).

Now in order to access the **TLIB** version of the **AWIDE** profile, the program that you are debugging must have an **//ISPTLIB** allocation to the appropriate ISPF Table Library. For debugging in TSO, that's usually automatic. For debugging batch jobs, however, you will need to add an **//ISPTLIB** DD card to your job's JCL. (This is true even when connecting to the debugging session from a TSO session!) For more information about ISPF-style **TLIBs,** see **HELP PROFILES TLIBPROFILES.**

Help PRofiles Factorydefaults C80

This Factory Default profile is named **C80.** It is suitable for use by C programmers using 80-column wide terminals. It is packaged as a hard-coded template built into Z/XDC's Assembler source code. It can be loaded by the **PROFILE RESET C80** command. See **HELP COMMANDS PROFILE RESET** for details.

Of the several dozen settings found in each profile, only a few differ from one Factory Default profile to the next. For those settings that are the same across each profile, see the several topics that follow **HELP PROFILES MENU.** For those settings that differ, see below.

Display Width

This **C80** profile sets **SET FORMAT NARROW.** This causes Z/XDC to format most of its displays to fit within 80 columns.

Display Layout

All Factory Default profiles divide the terminal's display area into two windows. The top window is the **Primary Window,** while the second window is a **Watch Window.** (For more information, see **HELP FULLSCREEN WINDOWS.)**

The Factory Default profiles differ between wide and narrow displays in the definition of the Watch Window:

- For the **C80** and **A80** profiles,
  - the Watch Window command line contains: **L PSW;L EPSW;L REGS.**
  - The window is 7 rows high.

- For the **CWIDE** and **AWIDE** profiles,
  - the Watch Window command line contains: **L PSWE;L EPSWE;L BEA;;L RWREGS;;L AREGS**
  - The window is 11 rows high.

Latent Commands String

The Factory Default profiles all set the same Latent Commands String: **L PSWE;L BEA;L RWREGS.** For more information about Latent Command Strings, see **HELP FULLSCREEN LOGGING LATENTCOMMANDS.**

STEP'ing PF keys

For all Factory Default profiles, most PF keys are defined identically. (See **HELP FULLSCREEN PFKEYS DEFAULTKEYS** for details.) Only the following keys differ:

9th - (F9) **STEP IN**
This is one of three PF keys that assist in stepping through C program execution. This one allows you to step through the execution of your program exactly one C statement at a time. If a subroutine call or function invocation is encountered, stepping will continue into the subroutine or function.

10th - (F10) **STEP OUT**
This is one of three PF keys that assist in stepping through C program execution. If execution is within a subroutine or function, This PF key allows you to let the rest of the routine run to completion and to recapture execution at the next statement past the statement that called the subroutine or invoked the function.

11th - (F11) **STEP OVER**
This is one of three PF keys that assist in stepping through C program execution. This PF key allows you to step through C program execution statement by statement, just like **STEP IN,** except that when you come upon a statement that calls a subroutine or invokes a function, that subroutine or function is allowed to run, and execution is recaptured when the subroutine or function returns.

Creating a Local Version of C80

If a Data Center wishes to make changes to this profile, a SysProg can load it (via the **PROFILE RESET C80** command) and make the desired changes and then save it via a **PROFILE SAVE C80 TFS** command. This creates a member named **TFSC80** in the SysProg's personal profile library.

To make this profile available System wide, the SysProg must then **move** it (not copy it) over to a suitable **//ISPTLIB** library. (The move must be done outside of Z/XDC.)

Then to load the modified profile, a programmer must use a **PROFILE READ C80** command (not PROFILE RESET C80).

Now in order to access the **TLIB** version of the C80 profile, the program that you are debugging must have an **//ISPTLIB** allocation to the appropriate ISPF Table Library. For debugging in TSO, that's usually automatic. For debugging batch jobs, however, you will need to add an **//ISPTLIB** DD card to your job's JCL. (This is true even when connecting to the debugging session from a TSO session!) For more information about ISPF-style **TLIBs,** see HELP PROFILES TLIBPROFILES.

Help PRofiles Factorydefaults CWide

This Factory Default profile is named **CWIDE.** It is suitable for use by C programmers using terminals defined as having **136** columns or more. It is packaged as a hard-coded template built into Z/XDC's Assembler source code. It can be loaded by the **PROFILE RESET CWIDE** command. See **HELP COMMANDS PROFILE RESET** for details.

Of the several dozen settings found in each profile, only a few differ from one Factory Default profile to the next. For those settings that are the same across each profile, see the several topics that follow **HELP PROFILES MENU.** For those settings that differ, see below.

Display Width

This **CWIDE** profile sets **SET FORMAT WIDE.** This causes Z/XDC to format most of its displays to fit within 136 columns.

Display Layout

All Factory Default profiles divide the terminal's display area into two windows. The top window is the **Primary Window,** while the second window is a **Watch Window.** (For more information, see **HELP FULLSCREEN WINDOWS.)**

The Factory Default profiles differ between wide and narrow displays in the definition of the Watch Window:

- For the **C80** and **A80** profiles,
  - the Watch Window command line contains: **L PSW;L EPSW;L REGS.**
  - The window is 7 rows high.

- For the **CWIDE** and **AWIDE** profiles,
  - the Watch Window command line contains: **L PSWE;L EPSWE;L BEA;;L RWREGS;;L AREGS**
  - The window is 11 rows high.

Latent Commands String

The Factory Default profiles all set the same Latent Commands String: **L PSWE;L BEA;L RWREGS.** For more information about Latent Command Strings, see **HELP FULLSCREEN LOGGING LATENTCOMMANDS.**

STEP'ing PF keys

For all Factory Default profiles, most PF keys are defined identically. (See **HELP FULLSCREEN PFKEYS DEFAULTKEYS** for details.) Only the following keys differ:

9th - (F9) **STEP IN**
This is one of three PF keys that assist in stepping through C program execution. This one allows you to step through the execution of your program exactly one C statement at a time. If a subroutine call or function invocation is encountered, stepping will continue into the subroutine or function.

10th - (F10) **STEP OUT**
This is one of three PF keys that assist in stepping through C program execution. If execution is within a subroutine or function, This PF key allows you to let the rest of the routine run to completion and to recapture execution at the next statement past the statement that called the subroutine or invoked the function.

11th - (F11) **STEP OVER**
This is one of three PF keys that assist in stepping through C program execution. This PF key allows you to step through C program execution statement by statement, just like **STEP IN,** except that when you come upon a statement that calls a subroutine or invokes a function, that subroutine or function is allowed to run, and execution is recaptured when the subroutine or function returns.

Creating a Local Version of CWIDE

If a Data Center wishes to make changes to this profile, a SysProg can load it (via the **PROFILE RESET CWIDE** command) and make the desired changes and then save it via a **PROFILE SAVE CWIDE TFS** command. This creates a member named **TFSCWIDE** in the SysProg's personal profile library.

To make this profile available System wide, the SysProg must then **move** it (not copy it) over to a suitable **//ISPTLIB** library. (The move must be done outside of Z/XDC.)

Then to load the modified profile, a programmer must use a **PROFILE READ CWIDE** command (not PROFILE RESET CWIDE).

Now in order to access the **TLIB** version of the CWIDE profile, the program that you are debugging must have an **//ISPTLIB** allocation to the appropriate ISPF Table Library. For debugging in TSO, that's usually automatic. For debugging batch jobs, however, you will need to add an **//ISPTLIB** DD card to your job's JCL. (This is true even when connecting to the debugging session from a TSO session!) For more

information about ISPF-style **TLIBs,** see HELP PROFILES TLIBPROFILES.

Help PRofiles Oldprofiles

Whenever Z/XDC reads a profile that was created by any older release, Z/XDC automatically converts that profile to the format required by Z/XDC's current release. The converted profile is not, however, automatically saved back to disk. A converted profile will not be saved until you explicitly issue a "PROFILE SAVE" or "PROFILE SAVE name" command.

Generally, when a converted profile is rewritten to disk, the original profile is overwritten. However, for default profiles this is not the case because the default profile's library member name changes with each release of Z/XDC. For example, if an X2.1 default profile was read from profile library member **XDCATT21,** converted, and resaved, it will be written to member **XDCDPV31.** Thus, the older default profile remains available in case you want to run the older Z/XDC again.

Once a profile has been converted and saved to disk, it can no longer be used by an older Z/XDC. Therefore, if you use a named profile (e.g. "PROFILE READ WAX58"), and if you might want to run an older Z/XDC again, you have to be careful not to overwrite the original profile library member. You can avoid doing this as follows: Whenever you read an older profile, immediately resave it under a new name. Example:

```
   PROFILE READ WAX58
   PROFILE SAVE WAX59
```

As indicated, above, this is not a problem for default profiles.

Help PRofiles DDnames

When Z/XDC needs to read a profile, it constructs a library member name (as described in HELP PROFILE PROFNAMES), and then it searches libraries allocated to the following ddnames until it finds that name:

```
   xxxPROF   (where xxx matches the the current Z/XDC's name)
   ISPPROF   (ISPF's standard profile library)
   xxxTLIB   (a TLIB)
   ISPTUSR   (a TLIB)
   ISPTLIB   (ISPF's standard TLIB)
```

Of course, only those ddnames that are actually allocated are searched.

If the profile is not found, then Z/XDC substitutes the characters "TFS" for the first three characters of the constructed member name, and then searches the same list of libraries again.

When Z/XDC needs to **write** a profile, it searches your debugging session allocations looking for one of the following ddnames:

```
   xxxPROF   (note, xxx usually is XDC)
   ISPPROF
```

Z/XDC looks for these ddnames in the order listed.

When Z/XDC finds one of these ddnames, it constructs the appropriate profile member name, and then it writes that member to the library.

Help PRofiles Tlibprofiles

Z/XDC writes session profiles to profile libraries. It **reads** profiles from either profile libraries (such as ISPPROF) or table libraries (such as ISPTLIB).

Generally, each individual TSO user has his own private profile library, while **all** users share a single set of table libraries. Accordingly, profile libraries usually have data representing each individual user's personal preferences, while table libraries contain system-wide default values for those preferences.

Z/XDC conforms to this structure. Individual personal profiles are located in personal profile libraries in members whose names start with **xxx** (where **xxx** matches the current Z/XDC's clone name). System-wide default profiles are located in table libraries in members whose names start with "TFS". They can be loaded explicitly via the command: **PROFILE READ name TFS.**

Your Data Center's default profile will be loaded automatically if you have not yet created your own personal profile. You can load it explicitly with the command: **PROFILE READ XDC TFS.**

(You can load one of Z/XDC's factory default profiles with the **PROFILE RESET [name]** command.)

Creating a System Wide Default Profile

If you wish to create a system-wide default profile, and if you have "write access" to suitable table libraries, then proceed as follows:

- Decide whether you want to create "the" default profile (see HELP PROFILE DEFAULTPROFILE) or a "named" default profile.
- Start an arbitrary Z/XDC debugging session. (Typing **XDCCALL IEFBR14** from ISPF's option 6 will suffice.)
- If the profile that you wish to create is similar to an existing profile, then use the **PROFILE READ name** command to load the existing profile into Z/XDC.
- Type **PROFILE** (without operands) to enter Z/XDC's "Profile Menuing System", and then display and change the settings as you see fit.
- When you are done, use the **QUIT** command to terminate the Profile Menuing System and return to Z/XDC's normal debugging display.
- If you wish to define a default MAPLIB list:
  - Use the **SET MAPLIBS dsn dsn ...** command to build an active MAPLIB list.
  - Use the **SET MAPLIBS SAVE** command to save the active MAPLIB list into storage-resident profile data as the default MAPLIB list.
  - Or use the **SET MAPLIBS SAVE name** command command to save the active MAPLIB list into storage-resident profile data as a named MAPLIB list.
- If you are creating a named default profile, then type **PROFILE SAVE name**

TFS. This will create in your profile library a member named "TFSname".

- Otherwise, if you are creating "the" default profile, then type **PROFILE SAVE XDC TFS** instead. This will create in your profile library a member named **TFSDPV31.**

- Use Z/XDC's **END COMPLETELY** command (factory default F4 or shift-F4) to terminate the Z/XDC debugging session.

- Go to ISPF's Move/Copy Utility (option 3.3) and move the member that you just now created from your profile library (usually ddname ISPPROF) to the appropriate table library (usually ddname ISPTLIB).

The new profile will now be available to all Z/XDC users on your system.

Help PRofiles Wide

If you do not routinely use large display geometries, I would strongly recommend that you start doing so! The improvement in your productivity, both inside and outside of Z/XDC, will be that significant!

If you do use large display geometries, then for Z/XDC you will want to establish a session profile that will let you take advantage of the extra real estate. As it happens, two of Z/XDC's **Factory Default** profiles are specifically designed to support large screen geometries. They are:

- **AWIDE:** This is suitable for Assembler programmers.
- **CWIDE:** This is suitable for C programmers.

For more information, see HELP PROFILES FACTORYDEFAULTS.

The Factory Default profiles are distributed in two forms:

- As hard-coded values that can be loaded with the **PROFILE RESET [name]** command.
- As members of an ISPF-style Table Library **(hlq.XDCV31.XDCTLIB)** that can be loaded with the **PROFILE READ [name]** command.

The difference is the Table Library members can be customized to local Data Center needs, while the hard-coded versions cannot. For more information, see HELP PROFILES TLIBPROFILES.

To load one of the xWIDE profiles and make it your personal default, proceed as follows:

First, decide whether you want to load the Assembler version **(AWIDE)** or the C version **(CWIDE).** For the purposes of this example, I will assume that you want to load AWIDE. So now issue the following commands:

PROF READ AWIDE TFS
This finds and loads a profile named **AWIDE** from a PDS member named **TFSAWIDE.** (Probably TFSAWIDE will be found in one of your ISPF Table Library, //xxxTLIB).

If you don't have access to a Table Library, you can instead load from the hard-coded copies of the Factory Defaults. In this case, you would issue the command **PROFILE RESET AWIDE** (or CWIDE).

PROF SAVE XDC

This renames your profile to **XDC** and saves it as your permanent default personal profile. It will be saved in your ISPPROF library in the member named **xxxDPV31.**

LIST PROF
This shows detailed information about your profile's name, library, version etc.

PROF
This starts up the Profile Menuing System. This a series of panels that display and allow you to change almost all profiled settings. When you are done, use **F3** one or more times to exit the Profile Menuing System.

If you do make any changes that you want to make permanent, issue the **PROFILE SAVE XDC** command again to save your changes to you own personal profile library (//xxxPROF or //ISPPROF).

For information about large geometry display terminals, see HELP FULLSCREEN TERMINALS GEOMETRIES.

Help Rexx

Z/XDC has support for user-written REXX execs that permit users to write their own Z/XDC commands using the REXX programming language. This support is referred to as the **rexx/XDC Interface.**

The rexx/XDC Interface provides mechanisms by which user written REXX execs can receive control, call any of a variety of Z/XDC internal service routines (access state data, parse keywords and address expressions, read and zap storage, etc.), display messages via Z/XDC's display services, and return control to Z/XDC.

Compiled REXX

Normally, REXX execs are interpreted. however, you can place compiled REXX execs in CEXEC form in your REXX library. The REXX interpreter will recognize that the REXX exec has been compiled, and automatically invoke either the compiler runtime (if it has been installed) or the alternate library (which will decompile the CEXEC to its original source statements and interpret it)

Summary

To run REXX execs, you have to allocate your library of REXX execs (and/or compiled REXX execs) to ddname **//xxxREXX.** Then run your exec by issuing the command: **REXX membername operands,** where:

- **REXX** is Z/XDC's command for running REXX subcommands.
- **membername** is the name of the library member that contains the REXX exec that you want to run.
- And **operands** is whatever argument string (if any) that your REXX exec may require.
- **xxx** (in ddname **//xxxREXX)** must match Z/XDC's current clone name (usually **XDC,** see HELP XDCCLONES for details).

You can also use the implicit command to execute a REXX exec: **%membername operands**

rexx/XDC Interface Services

ENVIRONMENT: When running REXX execs, a **rexx/XDC Interface** is required. Z/XDC's REXX Interface provides support for both the implicitly requested and the explicitly requested creation/destruction of a rexx/XDC Interface. For more information, see HELP REXX ENVIRONMENT.

SAYOUTPUT: REXX requires the existence of a **default character output stream** for use by the SAY and TRACE instructions. The rexx/XDC Interface provides support for creating that stream either implicitly or explicitly. For more information, see HELP REXX SAYOUTPUT.

INPUT REXX sometimes needs to request input from the user. If this needs to be done then REXX will read from a particular input stream. The rexx/XDC Interface provides support for creating that stream either implicitly or explicitly. For more information, see HELP REXX INPUT.

XDCSERVICES: Z/XDC's REXX Interface provides **REXX callable interfaces** to several of Z/XDC's internal services. For more information, see HELP REXX XDCSERVICES.

USERFUNCTIONS: You can add your own **user written function package** to the rexx/XDC Interface. For details, see HELP REXX USERFUNCTIONS.

XDCCOMMANDS: Currently, Z/XDC has only limited support by which user written execs can create and send commands to Z/XDC for execution. For more information, see HELP REXX XDCCOMMANDS.

SAMPLEEXECS: The Z/XDC distribution package contains several sample execs that illustrate how to use all of the built-in REXX functions that Z/XDC has. For more information, see HELP REXX SAMPLEEXECS.

To get to the above topics, type an **H** at the left above to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

Help Rexx Environment

A **REXX environment** is a structure of control blocks that is necessary for the running of REXX execs. This structure is built for REXX by IBM's IRXINIT routine and is taken down by IBM's IRXTERM routine.

An **XDC Environment** is a support mechanism by which REXX's **ADDRESS** instruction can be used to "send commands to the Environment" for execution. The rexx/XDC Interface, however, does not have usable support for an "XDC Environment". If you attempt to issue an **ADDRESS XDC xdccommand** instruction, the "XDC Environment" will issue an error message (DBC414E) and then terminate with RC=8.

So if you wish to issue Z/XDC commands, you will not be able to do so via REXX's **ADDRESS** instruction. Instead, you can use the **XDCCMD** built-in function. See HELP REXX XDCSERVICES XDCCMD for more information.

In an attempt to achieve clarity, I will use the term **XDC Environment** to refer solely to support for REXX's **ADDRESS XDC xdccommand** instruction. When referring to the environment of REXX control structures built by Z/XDC's **REXX STARTREXX** command, I will call that the **rexx/XDC Interface.**

Anyway, to continue...

The REXX command calls IRXINIT whenever it is necessary to build a rexx/XDC Interface. For more information about REXX interfaces and environments, see IBM's **REXX Reference** (SA22-7790).

Explicitly Managing the rexx/XDC Interface

The rexx/XDC Interface provides three built-in **execs** by which you can manage the REXX interface. They are:

REXX STARTREXX
This exec **explicitly** builds or rebuilds a rexx/XDC Interface. If a rexx/XDC Interface already exists, Z/XDC will call IRXTERM to take it down. Then it will call IRXINIT to build a new Interface.

REXX ENDREXX
This exec takes down a rexx/XDC Interface (if one exists) by calling the IRXTERM routine. If no Interface currently exists, no action is taken. In any case, the ENDREXX exec always completes normally.

REXX LISTREXX
This exec displays information about the current rexx/XDC Interface (if any). Note, Z/XDC's **LIST REXX** command produces an identical display.

Implicitly Managing the rexx/XDC Interface

If a **REXX execname** command is issued prior to a rexx/XDC Interface being created, then the **REXX** command will automatically call IRXINIT to build one.

If Z/XDC's **END** command (but not END KEEP) is issued while a rexx/XDC Interface exists, then IRXTERM will be automatically called to take down the Interface prior to the termination of the debugging session.

If the debugging session should end by any means other than an **END** command, then the rexx/XDC Interface (if any) is not automatically taken down.

Help Rexx SAYoutput

REXX writes output from the **SAY** instruction, tracing, and the **TE** command to an output stream.

That output stream can be a dataset or Z/XDC, dependent on settings established by the:

- XDCROUTE function
- SET REXX command

These settings can be changed at any time (including using the XDCROUTE function while a REXX procedure is executing) so the dataset is always allocated.

Note:
- When the output stream is routed to a ddname, and that ddname is allocated to the terminal, lines written to that ddname will result in a TSO line message being written to the terminal, and this will interrupt Z/XDC's fullscreen processing.
- When the output stream is routed to Z/XDC, Z/XDC may log the data, but no data is displayed until Z/XDC next sends output to the terminal, which may not happen until (at least) the REXX procedure terminates.

The use of a particular routing option thus depends on circumstances, but in general:
- If the REXX procedure is using tracing, the output is probably wanted immediately, so routing to a ddname is probably the best.
- If the REXX procedure is long-running, any SAY statement output will not be visible until the procedure terminates (at least). In this case routing to a ddname is probably the best.
- In other cases, routing to Z/XDC is recommended.

The following dataset allocations are used:

If the xxxTSPRT file (DDname) is allocated, it is used. (where xxx is the XDC clone name, normally XDC)

If the SYSTSPRT file (DDname) is allocated, it is used.

The xxxTSPTRT file (DDname) is allocated to SYSOUT and used. (where xxx is the XDC clone name, normally XDC)

Help Rexx Input

Sometimes REXX needs to solicit input from a user because the **[PARSE] PULL** instruction was used and the stack is empty or the **PARSE EXTERNAL** instruction was used. This input is solicited by REXX reading an input stream.

The input stream can be a dataset or Z/XDC, dependent on settings established by the:

- XDCROUTE function
- SET REXX command

These settings can be changed at any time (including using the XDCROUTE function while a REXX procedure is executing) so the dataset is always allocated.

Note:
- When the input stream is routed to a ddname, and that ddname is allocated to the terminal, the user can respond to the REXX procedure by typing the response and pressing the ENTER key.
- When the input stream is routed to Z/XDC, a prompt will be displayed that indicates the a REXX procedure requires input. The user can supply the input by responding to the displayed prompt and then pressing the ENTER key.

The following dataset allocations are used:

If the xxxTSIN file (DDname) is allocated, it is used. (where xxx is the XDC clone name, normally XDC)

If the SYSTSIN file (DDname) is allocated, it is used.

The xxxTSIN file (DDname) is allocated to the terminal and used. (where xxx is the XDC clone name, normally XDC) (this allocation is only done if Z/XDC is executing in a TSO address space)

Help Rexx XDCServices

Z/XDC's REXX interface contains a package of several built-in functions that serve as interfaces back to a selection of Z/XDC's internal services. These built-in functions can be called by user-written REXX execs. Below is a summary list of those functions. Collectively, these functions are referred to as **XDCccccc()** functions.

Most of the functions expect some of their input arguments to contain hexadecimal data or values. Also, some of the functions return hexadecimal data or values. In all cases, such hexadecimal values are presented in the form of character strings. In cases where the calling exec needs to manipulate the hexadecimal data numerically, it must first use REXX's **X2D()** function to convert the string to decimal. When done, the **D2X()** function can be used to convert the value back to a hexadecimal character string.

All of the **XDCccccc()** functions that return data will do so via stem variables, **not** via the function's return value. The returned data will include success/failure information. It also will include detailed XDC error messages when appropriate.

The default stem name for returned data will match the function's name; however, in all cases that default can be overridden via the function's input argument string.

For all cases, the function's return value (stored into RC, for example) will be the stem name (including the trailing dot) for the variables created by the function for returning its normal and error data.

Generally, whether a function succeeds, partially succeeds, or fails will be indicated by an error message returned in a variable named **stem.EMSG.** which will be set as follows:

- **Success:** stem.EMSG will be nulled.
- **Partial Success:** stem.EMSG will contain a descriptive information or warning message.
- **Failure:** stem.EMSG will contain a descriptive error message.

Notes about stem.EMSG:

All error messages start with the characters **DBC,** followed by three or four decimal digits, followed by a message type character (DBC###c or DBC####c).

The message type character will be one of the following:
- **I:** The message is informational.
- **W:** The message is a warning.
- **E:** The message describes an error.
- **S:** The message describes a severe error.

For more information, see HELP MESSAGES.

Subtopics Index

The following is a summary list of the built-in functions. For detailed descriptions, type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

- **XDCCMD(command,order,stem)**
  Stacks an XDC command for execution **after** the calling REXX exec completes.

- **XDCDUMP(prefix)**
  Sends messages to REXX's standard output stream displaying the names of and values in all exposed REXX variables whose names start with a given string.

- **XDCGET(prompt,stem)**
  get input via Z/XDC.

- **XDCMEMRY(address,size,asid,alet,stem)**
  Returns up to 4096 bytes of storage to the calling REXX exec.

- **XDCPARSE(expression,stem)**
  Parses an address expression and returns the resolved location (space and address).

- **XDCRDISP()**
  Repaints the user's terminal display.

- **XDCROUTE(direction,mode,stem)**
  Controls the routing of REXX input and output.

- **XDCSAY(string,address,asid,alet,stem)**
  Displays a given message via Z/XDC's Display Management.

- **XDCSTATE(type,stem)**
  Returns either error level or retry level state data (PSW, registers, key-mask, etc.) to the calling REXX exec.

- **XDCVARS(prefix,stem)**
  Returns to the calling REXX exec a stem array containing the names of all exposed REXX variables whose names start with a given string.

- **XDCZAP(address,data,asid,alet,stem)**
  Zaps up to 256 bytes of given data into a specified location of storage in any accessible address/data space.

- **XDCZSTAT(name,data,stem)**
  Zaps data into a specified retry level register or into permitted portions of the retry level PSW.

Help Rexx XDCServices XDCCmd

XDCCMD() is a built-in REXX function that can be used to pass a command string to Z/XDC for execution **after** the calling exec completes execution. (Currently, there is no way whatsoever of executing a Z/XDC command interactively with the REXX exec. However, see HELP REXX XDCCOMMANDS for additional information.

The command string can be stacked either LIFO or FIFO with respect to other command strings already stacked.

Syntax:

rc = **XDCCMD(command,order,stem)**

command
This argument is required. It contains the command string to be stacked for execution. It may be up to 255 characters long. It may contain multiple commands separated from each other by semicolons (;).

order
This argument is optional. It controls whether the command string is stacked LIFO or FIFO with respect to other command strings already stacked. It must contain one of the following strings. (Abbreviations are not accepted):

LIFO
The command string is to be stacked ahead of all previously stacked (and not yet executed) command strings.

FIFO
The command string is to be stacked following all previously stacked command strings.

If **order** is omitted, then **FIFO** is assumed by default.

stem
This argument is optional. If given, then it provides the stem name to be used for returned data. If omitted, then **XDCCMD.** is used by default. The given name may or may not including a trailing dot. If it does not, then one will be supplied before being used. Also, the name will be upcased before being used.

Returned Data:

Upon completion, the following variables are returned to the calling exec:

rc
The function’s returned value will always be the stem name used for all the other variables returned by this function. The name will be upcased and include the trailing dot.

stem.EMSG
This variable may contain an error message string, a warning message string or a null value according to whether the function failed or succeeded.

For examples of using the XDCCMD function, see the **RXTSTCMD** member of the XDCSAMP library. (The factory default name of the library is hlq.XDCV31.XDCSAMP.)

Help Rexx XDCServices XDCDump

XDCDUMP() is a built-in REXX function that can be used to display the names of and values in all exposed variables whose names start with a given string. The starting string can be a stem name, but it need not be. It can be anything at all.

The display generated by this function is sent to REXX's default character output stream (via standard REXX SAY instructions).

(See HELP REXX XDCSERVICES XDCVARS for information about a function that returns the names of all exposed variables in a stem array so that they can be examined and processed by the calling function.)

Syntax:

rc = **XDCDUMP(string)**

string
This argument is optional. It provides the match string for selecting the exposed variables whose names are to be returned. Those variables are selected whose names (including the stem name, if any) start with an exact match to the given string. String may be any of the following:

- Null or omitted
  All exposed variables are displayed.

- A string ending with a period
  All exposed variables are returned whose names start with an exact match (including the period) to the given string. (This method can be used to determine the names of all exposed variables having a given stem.)

- Any string at all
  All exposed variables are returned whose names start with an exact match to (or exactly match) the given string.

Returned Data:

rc
The function's returned value will be one of the following completion codes:

0 - The function completed normally. One or more exposed variables was

displayed.

-1 - The function did not find any exposed variables to display. No variable was found that matched the given prefix string.

For examples of using the XDCDUMP function, see the **RXTSTDUM** member of the XDCSAMP library. (The factory default name of the library is hlq.XDCV31.XDCSAMP.)

Help Rexx XDCServices XDCGet

XDCGET() is a built-in REXX function that can be used to obtain an input string via Z/XDC.

Syntax:

rc = **XDCGET(prompt,stem)**

prompt
A prompt message that will be displayed by Z/XDC
If the prompt message is omitted or blank or null a standard message will be displayed.

stem
This argument is optional. If given, then it provides the stem name to be used for returned data. If omitted, then **XDCGET.** is used by default. The given name may or may not include a trailing dot. If it does not, then one will be supplied before being used. Also, the name will be upcased before being used.

Notes

The value is returned in the rexx variable stem.REPLY.

The REXX **PARSE VAR** instruction can be used to parse the reply.

The **XDCGET** function was introduced by Z/XDC maintenance. If an error occurs on an invocation of this function, Z/XDC maintenance may need to be applied.

For examples of using the XDCGET function, see the **RXTSTGET** member of the XDCSAMP library. (The factory default name of the library is hlq.XDCV31.XDCSAMP.)

Help Rexx XDCServices XDCMemry

XDCMEMRY() is a built-in REXX function that can be used to extract up to 4096 bytes of data from any accessible storage located in any accessible address space or data space. (If Z/XDC is running authorized, then storage can be extracted from any

address space in the system that System Security permits.)

Syntax:

rc = **XDCMEMRY(address,size,asid,alet,stem)**

address
This argument is required. It specifies the storage address from which data is to be extracted. Address must be a character string of from 1 to 16 hexadecimal digits. Leading zeros may be provided but are not necessary.

Address may **not** be an address expression. If you want to use an address expression, then you must first use the XDCPARSE() built-in function to convert it to an ASID and address.

size
This argument is optional. It specifies the number of bytes of storage to be extracted. If given, then it must be a decimal number ranging in value from 1 to 4096. If omitted, then 4 is used as the default value.

asid
This argument is optional. If given, then it provides the ASID of the address space from which the data is to be extracted. It must be a string containing from 1 to 4 hexadecimal digits. (Note, the XDCPARSE() function, when resolving an address expression, will return an ASID as one of its outputs.)

If this argument is omitted, then the extracted data will come from Z/XDC's current default address space (as established by the SET ASID command).

alet
This argument is optional. If given, then it provides the ALET by which the target address space or data space can be reached. When given, this argument must be a string containing from 1 to 8 hexadecimal digits. When zero-padded (on the left) out to 8 digits, the result must be a currently valid ALET for the current address space (PASN-AL) or task (DU-AL). (Note, the XDCPARSE() function, when resolving an address expression, will return an ALET as one of its outputs.)

If this argument is omitted (or the value is zero), then no ALET is used for the storage access attempt. If real memory is being accessed, any ALET supplied is ignored.

stem
This argument is optional. If given, then it provides the stem name to be used for returned data. If omitted, then **XDCMEMRY.** is used by default. The given name may or may not including a trailing dot. If it does not, then one will be supplied before being used. Also, the name will be upcased before being used.

Returned Data:

Upon completion, the following variables are returned to the calling exec:

rc

The function's returned value will always be the stem name used for all the other variables returned by this function. The name will be upcased and include the trailing dot.

stem.EMSG
This variable may contain an error message string, a warning message string or a null value according to whether the function failed or succeeded.

stem.MEM
This variable will either be null, or it will contain either part of or all of the requested data from storage:

- If all of the requested data was extracted successfully, then:
  - **stem.MEM** will contain that data.
  - **stem.EMSG** will be null.

- If only part of the requested data was successfully extracted, then:
  - **stem.MEM** will contain that data.
  - **stem.EMSG** will contain an error message (probably DBC045E reporting that an 0C4 occurred).

- If none of the requested data was successfully extracted, then:
  - **stem.MEM** will be null.
  - **stem.EMSG** will contain an error message.

In all cases:
- The data within stem.MEM will be a character string consisting of an even number of hex digits.
- The value of **LENGTH(stem.MEM)** will be **twice** the number of bytes extracted.

For examples of using the XDCMEMRY function, see the **RXTSTMEM** member of the XDCSAMP library. (The factory default name of the library is hlq.XDCV31.XDCSAMP.)

Help Rexx XDCServices XDCParse

XDCPARSE() is a built-in REXX function that can be used to parse a given address expression. The parse is performed according to the syntax described in HELP ADDRESSING. The results include a storage address, an ASID and an ALET.

Syntax:

rc = **XDCPARSE(addressexpression,stem)**

addressexpression
This argument is required. It is a string containing the address expression to be parsed.

stem
This argument is optional. If given, then it provides the stem name to be used for returned data. If omitted, then **XDCPARSE.** is used by default. The given name

may or may not including a trailing dot. If it does not, then one will be supplied before being used. Also, the name will be upcased before being used.

Returned Data:

Upon completion, the following variables are returned to the calling exec:

rc
The function's returned value will always be the stem name used for all the other variables returned by this function. The name will be upcased and include the trailing dot.

stem.EMSG
This variable may contain an error message string, a warning message string or a null value according to whether the function failed or succeeded.

stem.ERROROFFSET
If the parse failed, then this variable will contain the offset (0-origin) of the parse cursor from the start of the address expression at which the parse error was discovered. (Also, stem.EMSG will contain an error message.)

If the parse succeeded, then this variable will contain -1. (And stem.EMSG will be null.)

stem.EXPRESSION
This variable will contain a copy of the expression that was parsed.

stem.ADDRESS
If the parse succeeded, then this variable will contain the resolved storage address. This address will be a 16-digit character string consisting entirely of hexadecimal digits.

If the parse failed, then this variable will contain 16 zeros.

stem.ASID
stem.ALET
If the parse succeeded, and if the resolved address is located within an address space, then:
- **stem.ASID** will contain the ASID of that space (a 4-digit character string consisting entirely of hexadecimal digits).
- **stem.ALET** will contain 8 zeros.

If the parse succeeded, and if the resolved address is located within a data space, then:
- **stem.ASID** will contain the 4-digit ASID of the address space that owns the data space.
- **stem.ALET** will contain the 8-digit ALET by which that data space can be accessed.

If the parse failed,then:
- **stem.ASID** will contain 4 zeros.
- **stem.ALET** will contain 8 zeros.

For examples of using the XDCPARSE function, see the **RXTSTPAR** member of the XDCSAMP library. (The factory default name of the library is hlq.XDCV31.XDCSAMP.)

Help Rexx XDCServices XDCRDisp

XDCRDISP() is a built-in REXX function that can be used to cause Z/XDC to fully repaint the user's terminal display. This function should be used if the calling REXX exec has caused messages to be sent to the display without Z/XDC being aware of it.

Syntax:

rc = **XDCRDISP()**

This function does not accept any arguments.

Returned Data:

Upon completion, the following variable is returned to the calling exec:

rc
The function's returned value will always be set to 0.

For examples of using the XDCRDISP function, see the **RXTSTRDI** member of the XDCSAMP library. (The factory default name of the library is hlq.XDCV31.XDCSAMP.)

Help Rexx XDCServices XDCROute

XDCROUTE() is a built-in REXX function that can be used to control the routing of REXX procedure input and/or output.

Syntax:

returnvalue = **XDCROUTE(direction,route,stem)**

returnvalue
The function return value, which is always the stem name (including the trailing '.') used by the function to return values and any error indication.
If no overriding stem name is used (as described below) the value will be **XDCROUTE.** (including the trailing '.').

direction,
The direction to control.

One of the following options:
IO To set or change both input and output routing.
INPUT To set or change just input routing
OUTPUT To set or change just output routing

route,
Where the input is from or/and the output is to.
One of the following options:
DD To set the input and/or output routing to/from a ddname.
XDC To set the input and/or output routing to/from XDC.
BOTH To set the input routing to XDC and the output routing to both a ddname and XDC.
SET Indicates that the settings from the 'SET REXX' command control the routing.
CURRENT Indicates that the current settings for routing are to be returned and no change is to be made.

stem
This argument is optional. If given, then it provides the stem name to be used for returned data. If omitted, then **XDCROUTE.** is used by default. The given name may or may not include a trailing dot. If it does not, then one will be supplied before being used. Also, the name will be upcased before being used.

Notes

The current settings (before any change) of the directions is always returned in stem.PxR (where X is I or O)

The new settings (after any change) of the directions is always returned in stem.CxR (where X is I or O) (these values may reflect the fact that **SET REXX ... FORCE** commands have been done to force a particular routing, regardless of the use of the XDCROUTE function)

The current settings of the the **SET REXX** command FORCE operand for the relevant directions is always returned in stem.CxF (where x is I or O) (It will have a value of YES or NO)

If **SET REXX IO=... FORCE** or **SET REXX INPUT=... FORCE** is in effect, any input routing options are processed and set but are effectively ignored.

If **SET REXX IO=... FORCE** or **SET REXX OUTPUT=... FORCE** is in effect, any output routing options are processed and set but are effectively ignored.

The **XDCROUTE** function was introduced by Z/XDC maintenance. If an error occurs on an invocation of this function, Z/XDC maintenance may need to be applied.

When the output stream is directed to a ddname, and that ddname is allocated to the terminal, lines written to that ddname will result in a TSO line message being written to the terminal, and this will interrupt Z/XDC's fullscreen processing.

If input and/or output is routed to a DD, records are truncated at the record length or 255 bytes, whichever is less.

When the output stream is directed to Z/XDC, Z/XDC may log the lines, but no lines are displayed until Z/XDC next sends output to the terminal, which may not happen

until (at least) the REXX procedure terminates.

For examples of using the XDCROUTE function, see the **RXTSTRTE** member of the XDCSAMP library. (The factory default name of the library is hlq.XDCV31.XDCSAMP.)

Help Rexx XDCServices XDCSAy

XDCSAY() is a built-in REXX function that can be used to pass a message to Z/XDC for display at the user's terminal. The message will be displayed within whatever window was current when the calling exec was invoked from Z/XDC.

Z/XDC may log the message, but no messages are displayed until Z/XDC next sends output to the terminal, which may not happen until (at least) the REXX procedure terminates.

The XDCSAY() function supports optionally associating a storage location with the message. When this is done, a Shortcut Entry Field is created by which the user can issue shortcuts (F D M Q etc.) for displaying or otherwise using the associated location. (See HELP SHORTCUTCMDS for more information.)

Syntax:

rc = **XDCSAY(string,address,asid,alet,stem)**

string
This argument is optional. It contains the message to be displayed. It may be any length up to approximately 800 characters long. If it is null or omitted, then a blank line is displayed.

address
asid
alet
These arguments are optional. If any are given, then they collectively define a storage location to be associated with the message. If they all are omitted, then no location is associated, and the message is displayed without a Shortcut Entry Field.

address
This argument specifies the storage address to be associated with the message. Address must be a character string of from 1 to 16 hexadecimal digits. Leading zeros may be provided but are not necessary. If address is omitted (and either asid and/or alet is given), then location 0 defaulted.

Address may **not** be an address expression. If you want to use an address expression, then you must first use the XDCPARSE() built-in function to convert it to an ASID and address.

asid

This argument provides the ASID of the address space to be associated with the message. It must be a string containing from 1 to 4 hexadecimal digits. (Note, the XDCPARSE() function, when resolving an address expression, will return an ASID as one of its outputs.)

If this argument is omitted (and either address and/or alet is given), then Z/XDC's current default address space (as established by the SET ASID command) will be associated.

alet
This argument provides the ALET by which the associated address space or data space can be reached. When given, this argument must be a string containing from 1 to 8 hexadecimal digits. When zero-padded (on the left) out to 8 digits, the result must be a currently valid ALET. (Note, the XDCPARSE() function, when resolving an address expression, will return an ALET as one of its outputs.)

If this argument is omitted then no ALET is used for the association.

stem
This argument is optional. If given, then it provides the stem name to be used for returned data. If omitted, then **XDCSAY.** is used by default. The given name may or may not including a trailing dot. If it does not, then one will be supplied before being used. Also, the name will be upcased before being used.

Returned Data:

Upon completion, the following variables are returned to the calling exec:

rc
The function's returned value will always be the stem name used for all the other variables returned by this function. The name will be upcased and include the trailing dot.

stem.EMSG
This variable may contain an error message string, a warning message string or a null value according to whether the function failed or succeeded.

For examples of using the XDCSAY function, see the **RXTSTSAY** member of the XDCSAMP library. (The factory default name of the library is hlq.XDCV31.XDCSAMP.)

Help Rexx XDCServices XDCSTate

XDCSTATE() is a built-in REXX function that can be used to retrieve either error level or retry level state information. (See HELP EXECUTIONLEVELS for more information about error level vs. retry level.) State information consists of such things as the PSW, register data, key-mask, etc.

Syntax:

rc = **XDCSTATE(type,stem)**

type
This argument is required. It controls whether error level or retry level state data is returned. It must contain one of the following strings. (Abbreviations are accepted):
ERROR
RETRY

stem
This argument is optional. If given, then it provides the stem name to be used for returned data. If omitted, then **XDCSTATE.** is used by default. The given name may or may not including a trailing dot. If it does not, then one will be supplied before being used. Also, the name will be upcased before being used.

Returned Data:

Upon completion, the following variables are returned to the calling exec:

rc
The function's returned value will always be the stem name used for all the other variables returned by this function. The name will be upcased and include the trailing dot.

stem.EMSG
This variable may contain an error message string, a warning message string or a null value according to whether the function failed or succeeded.

stem.SABEND
stem.UABEND
stem.REASON
These variables report what the ABEND is by which Z/XDC received control:

stem.SABEND: For system ABENDs, this variable will contain a 4-character string of hexadecimal digits reporting the ABEND code in the lo-order three digits. Notes:
- The hi-order digit of stem.SABEND will always be 0.
- If Z/XDC has received control as the result of a breakpoint, then stem.SABEND will contain "00C1".
- For user ABENDs, stem.SABEND will contain "0000".

stem.UABEND: For user ABENDs, this variable will contain the ABEND code as a decimal number. (For system ABENDs, stem.UABEND will be 0.)

stem.REASON: For all ABENDs, this variable will contain the ABEND reason code as reported by the RTM via the SDWACRC field. The reason code will be an 8-character string of hexadecimal digits.

Note, the values in these three variables are not affected by whether error level or retry level data is being returned.

stem.PIC
stem.TEAFLAGS
stem.TEA
stem.TEAREG
For program check ABENDs, these variables report additional information:

stem.PIC: This variable contains a 4-digit hexadecimal string reporting the Program Interrupt Code. These codes are as documented via the **z/Architecture Reference Summary** (SA22-7871)

stem.TEAFLAGS: This variable contains a 2-digit hex string whose values are flags that report whether or not stem.TEA contains valid data and, if so, then what kind of data it is. The bits within stem.TEAFLAGS's value have the following meanings:

- **X'40' (SDWATEAV):** stem.TEA contains an address.
- **X'20' (SDWATEIV):** stem.TEA contains an ASID.
- **X'08' (SDWATEPC):** stem.TEA contains a PC number.
- **Otherwise:** stem.TEA's contents are not valid.

Note, these flag values are mutually exclusive, so stem.TEAFLAGS can by analyzed by simple comparisons.

stem.TEA: This is the "Translation Exception Address" for s0C4 ABENDs and any other program check that might occur on an attempt to access storage. It may contain a storage address, an ASID or a PC number, depending upon the settings found in stem.TEAFLAGS. It's value is obtained from SDWATRAN or SDWATRNE, as appropriate. In all cases, the contents of this variable are a 16-digit hexadecimal string. The value may be any of the following:

- **Storage address**
  When stem.TEAFLAGS=X'40' (SDWATEAV), stem.TEA contains a storage address and some flags, as follows:
  - The hi-order 13 digits contain the address of the page of storage within which a storage access failed.
  - The lo-order order 2 digits (8 bits) contain flags that provide additional information about the storage access exception that occurred. These flags originated from the PSA's FLCTEAB3 field. They indicate whether the page tables used for the storage access were for a primary space reference, a secondary space reference, a home space reference or an access register qualified space reference. Refer to IBM's Data Control Blocks manuals for further details.
- **ASID**
  When stem.TEAFLAGS=X'20' (SDWATEIV), stem.TEA contains a 4-digit address space number, padded on the left with twelve zeros.
- **PC number**
  When stem.TEAFLAGS=X'08' (SDWATEPC), stem.TEA contains a 5-digit PC number, padded on the left with eleven zeros.

stem.TEAREG: When a translation exception occurs because of a problem with the contents of an access register, this variable contains the number (in decimal) of that register.

stem.PSW
This variable contains a string of 16 hexadecimal digits representing the contents of the 8-byte representation (the "scrunched" representation) of the error level or retry level Program Status Word. (The execution address is, of course, contained in the lo-order 31 bits of this PSW.)

stem.PKM
This variable contains a string of 4 hexadecimal digits representing the

contents of the 2-byte Program Key-Mask.

stem.ILC
For program check ABENDs, this variable contains the (decimal) length, in halfwords, by which the the corresponding execution address (retry level or error level) must be adjusted backwards to reach the start of the failing machine instruction. Notes:
- When the retry level and error level environments are the same, Z/XDC has already adjusted the retry level execution address. When that has occurred, the retry level stem.ILC will be set to zero to indicate that no further adjustment needs to be made.
- For non-program check ABENDs, instruction address adjustment is not needed; consequently, stem.ILC will be set to zero in such cases.
- The execution address is, of course, contained in the lo-order 31 bits of the PSW.

stem.AX
This variable contains a string of 4 hexadecimal digits representing the 2-byte error level or retry level Authority Index.

stem.EAX
This variable contains a string of 4 hexadecimal digits representing the 2-byte error level or retry level Extended Authority Index.

stem.LSE
This variable contains a string of 8 hexadecimal digits representing the 4-byte address of the descriptor field of the current linkage stack entry. This may be a state entry or a header entry, depending upon current circumstances.

stem.LSE0
This variable contains a string of 8 hexadecimal digits representing the 4-byte address of the descriptor field of the current linkage stack's first header entry (i.e., the start of the stack).

stem.xASID
stem.xIN
These variables contain 4-digit and 8-digit strings, respectively, representing the 2-byte and 4-byte Address Space Number and Address Space Instance Number, (respectively) of an error level or retry level address space. "x" may be either P, S, H or X, as follows:
- **P** indicates the Primary address space.
- **S** indicates the Secondary address space.
- **H** indicates the Home address space.
- **X** indicates the execution address space, i.e. the address space into which the PSW points. This will be either the primary space or the home space, depending upon the current Address Space Control Mode (ASC-MODE).

stem.Rn
This is a series of 16 variables who names range from stem.R0 through stem.R15. Each variable contains a string of 8 hexadecimal digits representing the 4 bytes of data found within the **low half** of corresponding error level or retry level general register.

stem.RHn
This is a series of 16 variables who names range from stem.RH0 through stem.RH15. Each variable contains a string of 8 hexadecimal digits representing the 4 bytes of data found within the **high half** of corresponding error level or

retry level general register.

stem.ARn
This is a series of 16 variables who names range from stem.AR0 through stem.AR15. Each variable contains a string of 8 hexadecimal digits representing the 4 bytes of data found within the corresponding error level or retry level access register.

stem.CRn
This is a series of 16 variables who names range from stem.CR0 through stem.CR15. Each variable contains a string of 8 hexadecimal digits representing the 4 bytes of data found within the **low half** of corresponding error level or retry level control register.

stem.CRHn
This is a series of 16 variables who names range from stem.CRH0 through stem.CRH15. Each variable contains a string of 8 hexadecimal digits representing the 4 bytes of data found within the **high half** of corresponding error level or retry level control register.

For examples of using the XDCSTATE function, see the **RXTSTSTA** member of the XDCSAMP library. (The factory default name of the library is hlq.XDCV31.XDCSAMP.)

Note, the **XDCVARS()** function can be used to return the names of all exposed variables having a given stem. Therefore, XDCVARS() is a good tool for reporting the names of all stem.variables created by the XDCSTATE() function.

Help Rexx XDCServices XDCVars

XDCVARS() is a built-in REXX function that can be used to return the names of all exposed variables whose names start with a given string. The string can be a stem name, but it need not be. It can be anything at all.

Syntax:

rc = **XDCVARS(string,stem)**

string
This argument is optional. It provides the match string for selecting the exposed variables whose names are to be returned. Those variables are selected whose names (including the stem name, if any) start with an exact match to the given string. String may be any of the following:

- Null or omitted
  (Almost) all exposed variables are returned. (Preexisting variables starting with the same stem name by which variable names are being returned are **not** returned.)

- A string ending with a period
  All exposed variables are returned whose names start with an exact match

(including the period) to the given string. (This method can be used to determine the names of all exposed variables having a given stem.)

- Any string at all
  All exposed variables are returned whose names start with an exact match to (or exactly match) the given string.

stem
This argument is optional. If given, then it provides the stem name to be used for returned data. If omitted, then **XDCVARS.** is used by default. The given name may or may not including a trailing dot. If it does not, then one will be supplied before being used. Also, the name will be upcased before being used.

Note, XDCVARS() will **never** return the names of any preexisting variables starting with the same stem name as the one given here.

Returned Data:

Upon completion, the following variables are returned to the calling exec:

rc
The function's returned value will always be the stem name used for all the other variables returned by this function. The name will be upcased and include the trailing dot.

stem.EMSG
This variable may contain an error message string, a warning message string or a null value according to whether the function failed or succeeded.

stem.0
This variable contains the number of names returned.

stem.nn
This is a series of variables, each containing a returned variable name. **nn** ranges from 1 up to the number given by stem.0. The names are returned all upcased and in alphabetic order.

For examples of using the XDCVARS function, see the **RXTSTVAR** and **RXTSTSTA** members of the XDCSAMP library. (The factory default name of the library is hlq.XDCV31.XDCSAMP.)

Help Rexx XDCServices XDCZAp

XDCZAP() is a built-in REXX function that can be used to zap data into up to 256 bytes of accessible storage.

When a zap occurs, the old contents of the target location are returned via a variable.

If Z/XDC is running authorized, then storage can be zapped pretty much anywhere that

System Security permits. This includes key 0 pages and store protected pages in any accessible address space or data space. It does not include locations protected by the System's "Low Address Protection Facility" (address space locations X'00000000'-X'000001FF' and X'00001000'-X'000011FF'.)

Syntax:

rc = **XDCZAP(address,data,asid,alet,stem)**

address
This argument is required. It specifies starting address of the storage to be zapped. The address must be a character string of from 1 to 16 hexadecimal digits. Leading zeros may be provided but are not necessary.

Address may **not** be an address expression. If you want to use an address expression, then you must first use the XDCPARSE() built-in function to convert it to an ASID and address.

data
This argument is required. It provides the data to be zapped into storage. It must be a string of hexadecimal digits. An even number of digits must be given. The number of digits may range from 2 to 512. The number of bytes zapped will be half the number of digits given.

asid

This argument is optional. If given, then it provides the ASID of the address space into which the data is to be zapped. It must be a string containing from 1 to 4 hexadecimal digits. (Note, the XDCPARSE() function, when resolving an address expression, will return an ASID as one of its outputs.)

If this argument is omitted, then the data will be zapped into Z/XDC's current default address space (as established by the SET ASID command).

alet

This argument is optional. If given, then it provides the ALET by which the target address space or data space can be reached. When given, this argument must be a string containing from 1 to 8 hexadecimal digits. When zero-padded (on the left) out to 8 digits, the result must be a currently valid ALET for the current address space (PASN-AL) or task (DU-AL). (Note, the XDCPARSE() function, when resolving an address expression, will return an ALET as one of its outputs.)

If this argument is omitted, then no ALET is used for the storage access attempt.

stem
This argument is optional. If given, then it provides the stem name to be used for returned data. If omitted, then **XDCZAP.** is used by default. The given name may or may not including a trailing dot. If it does not, then one will be supplied before being used. Also, the name will be upcased before being used.

Returned Data:

Upon completion, the following variables are returned to the calling exec:

rc
The function's returned value will always be the stem name used for all the other variables returned by this function. The name will be upcased and include the trailing dot.

stem.EMSG
This variable may contain an error message string, a warning message string or a null value according to whether the function failed or succeeded.

stem.OLD
When the zap completes successfully, this variable contains a copy of the target location's old data. The length of stem.OLD will be the same as the length of the data argument.

When the zap fails, stem.OLD will contain as much of the target location's old (and still current) data as can be read. The length of stem.OLD will be any even value ranging from 0 up to the length of the data argument (depending upon how much of the old data could be read successfully).

Note, when the zap fails, **none** of the zap data is stored. I.e. the zap target remains entirely unchanged.

For examples of using the XDCZAP function, see the **RXTSTZAP** member of the XDCSAMP library. (The factory default name of the library is hlq.XDCV31.XDCSAMP.)

Help Rexx XDCServices XDCZStat

XDCZSTAT() is a built-in REXX function that can be used to zap:
- Retry level general registers
- Retry level access registers
- Certain fields of the retry level PSW
- Certain other retry level state data

(See HELP EXECUTIONLEVELS for more information about error level vs. retry level.)

Syntax:

rc = **XDCZSTAT(name,data,stem)**

name
This argument is required. It identifies the object to be zapped. It must be one of the following:

- **ARn** An access register.
- **Rn** The low half (32 bits) of a general register.
- **RHn** The high half of a general register.
- **RWn** The entirety (all 64 bits) of a general register.

- **AMODE** The retry level addressing mode.

- **ASCMODE** The retry level address space control mode.
- **KEY** The retry level execution key.
- **RESUME** The retry level resume address.
- **STATE** The retry level execution state.

Notes:
- **n** (in the register names above) is a decimal number ranging from 0 to 15.
- All zappable registers and state values are part of the retry level environment. (Error level objects cannot be zapped.)
- See HELP EXECUTIONLEVELS for more information about error level vs. retry level.

data
This argument is required. It provides the data to be zapped. Its syntax varies according to the object to be zapped:

- For **ARn Rn** and **RHn**
  The data must be a string of hexadecimal digits that is **exactly** 8 digits long.

- For **RWn**
  The data must be a string of hexadecimal digits that is **exactly** 16 digits long.

- For **AMODE**
  The data specifies the desired addressing mode. It must be one of the following numbers:
  24
  31
  64

- For **ASCMODE**
  The data specifies the desired address space control mode. It must be one of the following keywords, (Abbreviations are permitted):
  PRIMARY
  SECONDARY
  HOME
  AR
  Note, SECONDARY and HOME modes can be set only when Z/XDC is running authorized.

- For **KEY**
  The data must be a decimal number ranging from 0 to 15. Note, When running non-authorized, only those execution key values can be set that are permitted by the retry level key-mask. (Usually, that will be only keys 8 and 9.) (When running authorized, any key value can be set regardless of the key-mask.)

- For **RESUME**
  The data must be a hexadecimal string, up to 8 digits long whose value ranges from X'00000000' to X'7FFFFFFF' (2**31-1). Note, data may **not** be an address expression. If you want to use an address expression, then you must first use the XDCPARSE() built-in function to convert it to an address.

- For **STATE**
  The data specifies the desired execution state. It must be one of the following keywords, (Abbreviations are permitted):

PROBLEM
SUPERVISOR
Note, SUPERVISOR state can be set only when Z/XDC is running authorized.

stem
This argument is optional. If given, then it provides the stem name to be used for returned data. If omitted, then **XDCZSTAT.** is used by default. The given name may or may not including a trailing dot. If it does not, then one will be supplied before being used. Also, the name will be upcased before being used.

Returned Data:

Upon completion, the following variables are returned to the calling exec:

rc
The function's returned value will always be the stem name used for all the other variables returned by this function. The name will be upcased and include the trailing dot.

stem.EMSG
This variable may contain an error message string, a warning message string or a null value according to whether the function failed or succeeded.

For examples of using the XDCZSTAT function, see the **RXTSTZST** member of the XDCSAMP library. (The factory default name of the library is hlq.XDCV31.XDCSAMP.)

Help Rexx Userfunctions

If you have your own, user written function package that you would like to make available for use by your REXX execs, Z/XDC's REXX Interface permits you to do that. During STARTREXX processing, while building the PACKTB control block (an input to IBM's IRXINIT routine), Z/XDC searches for the following load module names and builds PACKTB entries for all that it finds:

- **xxxFUSER:** user functions (But only if xxx <> XDC.)
- **XDCFUSER:** user functions
- **IRXFUSER:** user functions
- **xxxFLOC:** local functions (But only if xxx <> XDC.)
- **XDCFLOC:** local functions
- **IRXFLOC:** local functions
- **xxxEFMVS:** system functions (But only if xxx <> XDC.)
- **XDCEFMVS:** system functions (This is Z/XDC's distributed function package.)
- **IRXEFMVS:** system functions (This is MVS's standard function package.)
- **IRXEFPCK:** system functions (This is MVS's standard function package for TSO/ROSCOE environments.)

If you would like to provide your own function packages, you can do so by naming them any of the above names that are not already reserved for use by Z/XDC, by TSO or by MVS.

Note, the **LIST REXX** command reports information about what function packages are

loaded and what functions come from which packages.

Help Rexx XDCCommands

Z/XDC has only limited support by which a REXX exec can send commands to Z/XDC for execution. Currently, there is no way for a REXX exec to run commands interactively with Z/XDC. Z/XDC only has support by which commands sent to Z/XDC are queued for execution **after** the REXX exec completes.

Z/XDC does create a command environment named **XDC** (i.e. an **"XDC Environment");** however Z/XDC does not yet accept commands for execution through that environment. When a REXX exec attempts to send such a command, the XDC Environment only issues an error message (DBC414E), and then returns to the exec with the RC special variable set to 8. The command itself is discarded.

However, the rexx/XDC Interface does include a built-in REXX function named **XDCCMD()** by which a REXX exec can send commands to Z/XDC. But XDCCMD() only queues the commands for execution after the REXX exec itself completes. For more information, see HELP REXX XDCSERVICES XDCCMD.

Tip: When stacking commands for execution after the current exec completes, keep in mind, that one such command could be a **REXX execname** command. A technique such as this could be used to return control to your REXX routines after using other Z/XDC commands to set things up.

Help Rexx SAMpleexecs

The Z/XDC distribution package contains several sample execs that illustrate how to use all of the built-in REXX functions that Z/XDC has. All of the samples can be found in a library named hlq.XDCV31.XDCSAMP. (Note, this is the factory default name. Ask your Systems Programmer for the library's actual name at your Data Center.)

The sample execs are all in members whose names are RXTSTccc. When you have questions about how to use a particular built-in REXX function, after reading the appropriate Built-in Help topic (HELP REXX XDCSERVICES XDCccccc), then you should examine carefully the RXTSTccc sample exec that particularly illustrates how to use REXX function of interest.

Running the Sample REXX Execs

Running the sample execs is pretty easy. all you have to do is:

- Allocate the XDCSAMP library to ddname **xxxREXX.** Example: If your debugging session is running in TSO (i.e. you're not using cdf/XDC), then the following command can be issued either before starting up your debugging session or from within Z/XDC:

```
TSO ALLOCATE DDNAME(XDCREXX) DATASET('hlq.XDCV31.XDCSAMP)' SHR REUSE
```

- From within Z/XDC, issue **REXX RXTSTccc,** where RXTSTccc is one of the sample execs listed below.

The RXTSTccc Sample Execs

The following sample execs can be found in the XDCSAMP library:

RXTSTCMD: This exec illustrates various valid and invalid calls to the **XDCCMD()** built-in function. See HELP REXX XDCSERVICES XDCCMD for more information.

RXTSTDUM: This exec illustrates various calls to the **XDCDUMP()** built-in function. See HELP REXX XDCSERVICES XDCDUMP for more information.

RXTSTENV: This exec illustrates what happens to attempts to issue Z/XDC commands via the XDC Environment. (They display a DBC414E message, and then they abort with RC=8.)

This exec also issues a **LIST XDC** command (via the XDCCMD() function), so among other things, this exec can be used to definitively show to which xxx clone the rexx/XDC Interface belongs.

RXTSTMEM: This exec illustrates various valid and invalid calls to the **XDCMEMRY()** built-in function. See HELP REXX XDCSERVICES XDCMEMRY for more information.

RXTSTPAR: This exec illustrates various valid and invalid calls to the **XDCPARSE()** built-in function. See HELP REXX XDCSERVICES XDCPARSE for more information.

RXTSTRDI: This exec illustrates calls to the **XDCRDISP()** built-in function. See HELP REXX XDCSERVICES XDCRDISP for more information.

RXTSTRTE: This exec illustrates various calls to the **XDCROUTE()** built-in function. See HELP REXX XDCSERVICES XDCROUTE for more information.

RXTSTSAY: This exec illustrates various valid and invalid calls to the **XDCSAY()** built-in function. See HELP REXX XDCSERVICES XDCSAY for more information.

RXTSTSTA: This exec illustrates various valid and invalid calls to the **XDCSTATE()** built-in function. See HELP REXX XDCSERVICES XDCSTATE for more information.

RXTSTVAR: This exec illustrates various valid and invalid calls to the **XDCVARS()** built-in function. See HELP REXX XDCSERVICES XDCVARS for more information.

RXTSTZAP: This exec illustrates various valid and invalid calls to the **XDCZAP()** built-in function. See HELP REXX XDCSERVICES XDCZAP for more information.

RXTSTZST: This exec illustrates various valid and invalid calls to the **XDCZSTAT()** built-in function. See HELP REXX XDCSERVICES XDCZSTAT for more

information.

Help SEcurity

Z/XDC is a very powerful system development and debugging tool. When used properly, it can provide an enormous productivity boost to the development and debugging of a very wide range of customer programs and systems.

Because of its power, Z/XDC also can be quite dangerous. When Z/XDC is used to debug authorized programs and systems, a careless programmer can also quite easily cause the entire operating system to crash in less time than it takes the ENTER key to rise after being pressed! Accordingly, Z/XDC tries to take precautions to make it less likely for careless damage to occur.

But first, it is important to understand the true nature of the system security and integrity issues that are raised by Z/XDC. To begin with, when Z/XDC is running non-authorized:

- It **cannot** make itself or any other program authorized.
- It **cannot** alter any storage that the program being debugged cannot alter.
- It **cannot** do any damage that the program being debugged also couldn't do (if it were so coded).
- Z/XDC uses a private service SVC to perform certain functions that cannot be performed by non-authorized programs **(exactly** as IBM SVC routines do); but Z/XDC's service SVC does **not** make Z/XDC (or any other caller) authorized, and if it is misused, it nonetheless cannot be used to violate system integrity or security (just as IBM SVC routines also cannot be used to violate integrity or security).

When Z/XDC is running **authorized:**

- It **can** be used to look at the private areas of any address space in the System!
- It **can** be used to alter any storage in any address space in the System!
- It **can** be used to look at and alter common storage, PLPA storage, and real storage!

All of these capabilities can, to a certain extent, be controlled by system security rule definitions (described in the following sections); nonetheless, great care must be exercised in deciding who is and who is not allowed to run Z/XDC authorized!

Allowing a programmer to run Z/XDC authorized is (from a security point of view) not any different from allowing a programmer to write programs and place them into authorized libraries. There is not a thing that can be done with Z/XDC that cannot also be done with ASMA90 (the High Level Assembler). Authorized programs or subroutines that compromise security can be written in 20 lines of code or less! Once the knowledge is present, the effort is insignificant!

So, assume that a particular programmer has been allowed to run Z/XDC without security restrictions. What then are the security and integrity considerations? Well, a programmer might be trustworthy or careless or malicious (or various combinations).

- For the trustworthy programmer (who is not also careless), there is no security or integrity problem. The programmer will not violate security, and he will be sufficiently careful in his use of Z/XDC so as to crash the System only rarely, if ever.

- For the careless but otherwise trustworthy programmer, there is no security problem, but there is an integrity problem, because a careless programmer can easily use Z/XDC to damage the System. However, he can also just as easily code damaging routines in his own program. On the other hand, Z/XDC's symbolic displays can considerably assist a programmer's comprehension of his code, so the overall net effect can be beneficial. In any case, Z/XDC's security rules permit some refinements that might be useful for controlling such a programmer without at the same time excessively restraining him. It is up to management to decide whether or not the benefits of a careless programmer's output is worth the trouble caused in achieving that productivity.

- For the malicious programmer, the security issues raised by Z/XDC are very serious. A malicious programmer must **never** be allowed to run Z/XDC in an authorized environment for **exactly** the same reasons that such a programmer must never be given WRITE access to an authorized library! Z/XDC, like an Assembler, allows a programmer (through its ZAP command) to write or modify essentially arbitrary code, and such code (whether Assembler created or Z/XDC created), being authorized, could be used to subvert any security system. Therefore, if you would not allow a programmer to have WRITE access to an authorized library, then you also must not allow him to use Z/XDC authorized, **and vise-versa!** A programmer who is not allowed to run Z/XDC authorized also should not be allowed to have WRITE access to authorized libraries!

Distinguishing among programmers according to their security and integrity risks is, of course, a management problem, not a technical one. The choices of what to do once a classification has been made, is, on the other hand, influenced by the nature of the available security controls. Z/XDC offers the following controls:

- When a debugging session starts up authorized or becomes authorized, Z/XDC issues a security system call to see if the current programmer is permitted to run Z/XDC authorized. This is Z/XDC's major security defense. Programmers who are denied authorized access cannot use Z/XDC alone to violate system security or integrity. A non-authorized Z/XDC has no more capabilities in these areas than any other non-authorized program.

  On the other hand, once a programmer has been given **authorized access** to Z/XDC, **it is possible** for him to use Z/XDC **to subvert** all other security controls, **including** the ones described below!

- Whenever a programmer issues a Z/XDC command that alters storage, Z/XDC issues a security system call to see if the current programmer is permitted to alter the targeted storage. Z/XDC's security call distinguishes areas of storage according to load module name, csect name, subpool number, free space, etc. (storage altering commands are ZAP, POST, HOOK, TRACE, TRAP, AT, ATX, OFF, GETMAIN, FREEMAIN, LOAD module and DELETE module.)

- When a debugging session first enters Foreign Address Space Mode (to display another address space's private area), Z/XDC issues a security system call to see if the current programmer is permitted to view the targeted address space. See HELP VIRTMEM XDCACCESS FASM for more information about Foreign Address Space Mode.

- When a debugging session first uses Foreign Address Space Mode **to zap** another address space's private area, Z/XDC issues a security system call again to see if the current programmer is permitted to alter the targeted address space. Then for each alteration request, security is called again for permission to alter the targeted storage according to its location: load module, csect, subpool,

free area, etc.

- When the HOOK command is used to create (on the fly) a debugging session in another address space, Z/XDC issues a security system call to see if the current programmer is permitted to alter the targeted address space. See HELP COMMANDS HOOK for more information about the HOOK command.

- When Z/XDC is used to debug a space switching PC routine, Z/XDC issues a security call to see if the current programmer is permitted to alter the address space that owns that PC routine. (See HELP DEBUGGING PCROUTINES for more information.)

- When Z/XDC is used as an FRR, and the program being debugged had held locks, then in order to Z/XDC to process, those locks had to be released. Generally, if the user wishes to resume that program's execution, those locks will have to be reacquired. That is an **extremely dangerous** thing to do, so before permitting such a resumption to occur, Z/XDC issues a security call to see if the current programmer is permitted to do this. Only programmers expert in the workings of the Operating System should be given permission to perform such resumptions, but only on sandbox systems, **never** on a production system! (For more details, see HELP DEBUGGING LOSTLOCKS.)

- When a programmer uses cdf/XDC to connect to an existing debugging session in a background address space, Z/XDC issues a security system call to see if the current programmer is permitted both to view and to alter the targeted address space.

  It is important to realize that cdf/XDC can be used only to debug background jobs that have **already** been altered by a programmer who has had to perform the act of installing (or HOOK'ing) a Z/XDC interface into the background program. cdf/XDC **cannot** be used to connect to background programs that have not been so prepared. See HELP XDCSRVER CDF for more information.

For a discussion of the differences between Foreign Address Space Mode and cdf/XDC, see HELP XDCSRVER CDF FASM.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

RULES - General information about defining access control rules for Z/XDC in RACF, CA-ACF2, and CA-Top Secret.

AUTH - Controlling the use of Z/XDC running authorized using Z/XDC security rules.

CDF - Controlling access to background debugging sessions via cdf/XDC.

FASM - Controlling access to other address spaces via Z/XDC's Foreign Address Space Mode.

LOSTLOCKS - Controlling program resumption when locks have been lost.

ZAP - Controlling the the usage of Z/XDC commands to alter storage in both the local address space and any foreign address space.

GZAP - This is an obsolete rule that is no longer supported by Z/XDC. If your

Data Center is still using this rule, **then read this topic!**

ACIF - Providing a customer written security exit routine for Z/XDC.

CACHE - Information about an internal security cache that Z/XDC maintains to improve performance.

TRACE - A detailed description of a Z/XDC security trace that can be requested.

Help SEcurity Rules

Z/XDC supports the following z/OS security environments:
- RACF from IBM.
- CA-ACF2 from Computer Associates.
- CA-Top Secret also from Computer Associates.
- No security installed.

Of course when no security system is installed, Z/XDC runs without restrictions on the assumption that the customer has decided that he does not need security. Nonetheless, the customer may still write a Z/XDC specific exit routine if he wants to implement special security for Z/XDC. See HELP SECURITY ACIF for more information.

When a security system is installed, and a security sensitive event arises, Z/XDC constructs an appropriate rule string to represent that event, and then calls security to see whether or not the current programmer is permitted to cause the event. The security system may return any of three responses: PERMITTED, DENIED, or NOT PROTECTED.
- For PERMITTED and DENIED, Z/XDC will proceed with or abort the event as directed.
- For NOT PROTECTED, with the exception of one case, Z/XDC will **proceed** with the event! This has the following consequences:
  - Customers who are not concerned with security can install and begin using Z/XDC more quickly.
  - In order to secure usage of Z/XDC, concerned customers must either create Z/XDC related access control rules **prior** to installing Z/XDC, or install Z/XDC's load modules into libraries whose access is appropriately restricted by classic dataset access control rules.

  The exception is for the **XDC.LOSTLOCKS** rule. When that rule is not defined, Z/XDC will proceed as if **denied** had been returned. (This is of concern only to programmers who wish to use Z/XDC as an FRR routine. See HELP DEBUGGING LOSTLOCKS for more information.)

The rule strings that Z/XDC builds are discussed in detail in the following sections; however, the strings vary slightly according to which security system is installed:

- For RACF protected systems, all of Z/XDC's rule strings start with the characters "XDC.". Examples:
  ```
  XDC.FASM.JES2
  XDC.ZAP.COMMON.SUBPOOL.245
  XDC.AUTH
  ```
  Note, "XDC" is always used here, **never** a clone name. So unlike most other uses of Z/XDC's name, when Z/XDC has been renamed, the security rule strings that Z/XDC constructs still use "XDC", not the clone name. (For more information about XDC clone names, see HELP XDCCLONES.)

- For CA-ACF2 and CA-Top Secret protected systems, all of Z/XDC's rule strings

start **without** the characters "XDC.". Examples:
FASM.JES2
ZAP.COMMON.SUBPOOL.245
AUTH

Throughout this Built-in Help, whenever a security rule is discussed, it will be shown **with** the "XDC." prefix.

For further details about creating Z/XDC specific security rules, the Installer or Security Officer should refer to Z/XDC's **Installation Guide.** For detailed information about the rules themselves, please type HELP ∗NEXT (F11) (HELP SECURITY AUTH).

Help SEcurity AUth

When a debugging session starts up or becomes authorized, Z/XDC issues a security system call to see if the current programmer is permitted to run Z/XDC authorized. This is Z/XDC's major security defense. Programmers who are denied authorized access cannot use Z/XDC alone to violate system security or integrity. A non-authorized Z/XDC has no more capabilities in these areas than any other non-authorized program. On the other hand, once a programmer has been given authorized access to Z/XDC, it is possible for him to use Z/XDC to subvert all other security controls, **including those provided by Z/XDC itself!**

There are several ways to control whether or not Z/XDC is allowed to run authorized. The most flexible way is to create the Z/XDC specific security rule that is described below in this section. Other ways involve using classic dataset access control rules to limit who is and is not permitted to access Z/XDC's load libraries.

Z/XDC considers itself to be authorized when it can issue a MODESET macro and not be ABENDed by the System. The MODESET macro is the standard method for an authorized program to switch into "supervisor state". The MODESET macro will succeed when any of the following are true:

- The Job Step Control Block (JSCB) has JSCBAUTH=1 set. This is the normal "badge of authority" that is set by z/OS whenever it initiates an authorized program.

- The PSW has the "problem state" flag set off. This indicates that the program is running in "supervisor state".

- The PSW contains an execution key value in the range of 0 to 7. These are execution keys that z/OS reserves for use by supervisory programs. (Problem mode programs can theoretically use execution keys in the range of 8 to 15 (X'F'), but usually they are forced to use only key 8 or 9.)

- The PSW key-mask permits the SPKA instruction to set the execution key to values in the 0 through 7 range.

- The current Task Control Block (TCB) has TCBFSM=1 set. z/OS uses this flag for certain supervisory tasks that run in all address spaces.

The security call that Z/XDC makes has the following characteristics:

- The rule string that Z/XDC builds and passes to security is:

    - "XDC.AUTH" for RACF protected systems.
    - "AUTH" otherwise.
  - The access level that Z/XDC requests is READ.
  - The userid that is passed to security is, of course, the userid that security has already associated with the Home Address Space in which Z/XDC is running.

If the security system responds with DENIED, then Z/XDC immediately terminates and allows the current ABEND to percolate to the next older ABEND error recovery routine, if any. In other words the ABEND proceeds as if Z/XDC does not exist. (Note that Z/XDC breakpoints are, as far z/OS is concerned, nothing more than s0C1 ABENDs.)

If the System responds either with PERMITTED or NOT PROTECTED, then Z/XDC allows the debugging session to proceed normally.

Note, when Z/XDC is running under a clone name other than "XDC", in most instances where the name "XDC" appears as a part of another name (program names, ddnames, VTAM names, etc.), the clone name would be used instead of "XDC". However for security rules, this substitution does **not** occur! So in RACF protecting systems, the rule string constructed by Z/XDC remains XDC.AUTH, not clonename.AUTH. (For information about renaming Z/XDC, see HELP XDCCLONES.)

Help SEcurity CDf

When a programmer uses cdf/XDC to connect to an existing debugging session in a background address space, Z/XDC needs to determine whether or not to allow the connection. Z/XDC performs two checks:

- First, Z/XDC determines whether or not the targeted address space has an "ownerid" (a non-blank, non-asterisk value in the **ACEEUSRI** field of the address space's **ACEE** control block). If so, then it checks to see if the programmer's userid exactly matches that ownerid. If so, then Z/XDC permits the access without performing any further checks.

- If the Target Address Space either does not have an ownerid or its ownerid does not match the programmer's userid, then Z/XDC issues a security system call to see if the current programmer is permitted both to view and to alter the targeted address space. The security rules that Z/XDC checks are the **XDC.FASM.jobname** rules. **(Jobname** may contain wildcards.)

It is important to understand that cdf/XDC can be used only to debug background jobs that have **already** been altered by a programmer who has had to perform the act of installing (or **HOOK'ing)** a Z/XDC interface into the background program.

cdf/XDC **cannot** be used to connect to background programs that have not been so prepared. See **HELP XDCSRVER CDF** for more information.

(For a discussion of the differences between Foreign Address Space Mode and cdf/XDC, see **HELP XDCSRVER CDF FASM.)**

In order to use cdf/XDC to connect to a background debugging session, a programmer **first** has to logon to cdf/XDC. For a detailed discussion of this logon

process, see **HELP XDCSRVER CDF LOGON.** Briefly, however, there are two ways in which a logon can occur:

- If the programmer has already logged onto TSO, then he can use an ISPF panel (Z/XDC's Startup Panel) to request a connection to cdf/XDC. In this case, the logon to cdf/XDC is automatic: cdf/XDC determines from TSO what the programmer's userid is and accepts the connection without requiring the programmer to go through a separate logon process.

- If the programmer connects to cdf/XDC from an idle VTAM terminal (via **LOGON APPL=XDC),** then cdf/XDC first requires him to give his TSO userid and password before accepting the logon.

In either case, once cdf/XDC accepts the logon, it then knows who the programmer is.

cdf/XDC then has to develop a **Job Selection Menu** of background debugging sessions to which the programmer is allowed to connect. It does this by checking all pending debugging sessions (if any). For each pending session found, cdf/XDC determines whether or not the programmer is allowed to make a connection. If so, then the session's name will be displayed to the programmer. If not, then the session will not be displayed, and so the programmer will not even know that the session exists.

When the programmer makes his selection, the **same** access determination process is performed again just to make sure that the programmer's right to connect still exists. This is because any number of minutes or hours may have passed between the time that the selection list was displayed and the time that the programmer finally makes his selection.

In order to determine whether or not a programmer has the right to select and debug a particular background job, cdf/XDC performs the two checks described above (the ownerid check and the **XDC.FASM.jobname** check). If the ownerid check fails and cdf/XDC then has to perform the **XDC.FASM.jobname** check, then it performs that check as follows:

- The rule string that Z/XDC builds and passes to security is:
  - **XDC.FASM.jobname** for RACF protected systems.
  - **FASM.jobname** otherwise (ACF2 and TSS).

  where **jobname** is the name of the address space in which the target debugging session is running.

- The access level that Z/XDC requests is **UPDATE.**

- The userid that is passed to security is the (password confirmed) userid that cdf/XDC received from TSO or from the programmer when he made his connection to cdf/XDC.

If the security system responds with **DENIED,** then cdf/XDC omits the debugging session from the job selection list, and so the programmer has no way to connect to it.

If the System responds either with **PERMITTED** or **NOT PROTECTED,** then cdf/XDC adds the debugging session to the "job selection list" that is displayed to the programmer, and it permits the programmer to connect to the debugging session.

Note, when Z/XDC is running under a clone name other than **XDC,** then contrary to Z/XDC's practice in other situations, with security rules, Z/XDC does **not** substitute the clone name in place of **XDC.**

So in RACF protecting systems, the rule strings constructed by Z/XDC remain **XDC.FASM.jobname, *NOT*** clonename.FASM.jobname.

For information about renaming Z/XDC, see **HELP XDCCLONES.**

The rule string that Z/XDC builds for controlling cdf/XDC's access to an address space is the **same** string that it builds for controlling access via Foreign Address Space Mode. But please note the following:

- The requested access level for cdf/XDC connections is always **UPDATE;**
- Whereas, for **FASM** connections (Foreign Address Space Mode connections):
  - The initial check made is only for **READ** access.
  - **UPDATE** access (via **FASM)** is not checked for until the programmer issues a storage altering command **(ZAP, POST, GETMAIN, FREEMAIN, HOOK,** etc.).

See **HELP SECURITY FASM** for more information.

Help SEcurity Fasm

Z/XDC's "Foreign Address Space Mode" is a mode that permits a programmer to view and, perhaps, to alter the private area of another address space **without** requiring that Z/XDC be running in that other address space. Foreign Address Space Mode also can be used by a programmer to access real storage (i.e. not virtual storage). For more information, see HELP VIRTMEM XDCACCESS FASM.

For a discussion of the differences between Foreign Address Space Mode and cdf/XDC, see HELP XDCSRVER CDF FASM.

A programmer **cannot** use Foreign Address Space Mode unless Z/XDC is already running authorized (unlike the Cross Domain Facility which does not require authorization). This means that before a programmer can be permitted to use FASM mode, he must first be permitted to run Z/XDC authorized. See HELP SECURITY AUTH for more information.

Once a programmer has Z/XDC running authorized, all he then needs to do is use Z/XDC's SET ASID command to select the address space that he wishes to connect to. From then on, all of his FORMAT, LIST, DISPLAY, ZAP, HOOK, and POST commands are directed towards the selected address space. No cooperation from that address space is needed. For displays, the address space continues execution completely unaffected by the programmer's actions. For storage altering command (ZAP, HOOK, POST, TRAP, etc.) address space, of course, is directly affected by the programmer's actions. For more information, see HELP COMMANDS SET ASID.

A programmer can enter FASM mode also by using the ~ASID(...) built-in function within an address expression. For more information, see HELP FUNCTIONS ASID.

When a programmer first uses the SET ASID command or the ~ASID(...) built-in function to enter FASM mode against another address space, Z/XDC has to determine whether or not to permit access by the programmer to that address space. Z/XDC performs two checks:

- First, Z/XDC determines whether or not the targeted address space has an

"ownerid" (a non-blank, non-asterisk value in the ACEEUSRI field of the address space's ACEE control block). If so, then it checks to see if the address space from which the SET ASID command is being issued (usually the programmer's userid) exactly matches that ownerid. If so, then Z/XDC permits the access without performing any further checks.

- If the Target Address Space either does not have an ownerid or if the ownerids do not match, then Z/XDC issues a security system call to see if the current programmer is permitted to view the targeted address space. The security rules that Z/XDC checks are the "XDC.FASM.jobname" rules. ("Jobname" may contain wildcards.) The access level that Z/XDC requests is READ.

These security checks are made each time the programmer accesses an address space that he has not previously accessed during the debugging session.

When a programmer first uses a storage altering command against another address space, the security system may be called again to see if the programmer has UPDATE access to the targeted address space. If the user's access to the address space is based on an ownerid match, then there is no need to call security; Z/XDC just assumes that the alter-type access is OK. On the other hand, if the user's access is based upon the XDC.FASM.jobname rule, then Z/XDC calls security against that same rule to see if the programmer has UPDATE access to it and, therefore, to the targeted address space. This security call is made for each different address space that the programmer attempts to alter.

If Z/XDC makes a security call, then it does so as follows:
- The rule string that Z/XDC builds and passes to security is:
  - "XDC.FASM.jobname" for RACF protected systems.
  - "FASM.jobname" otherwise.

  where "jobname" is the name of the address space being targeted by the FASM connection. Special case: the rule string "XDC.FASM.*REAL*" is built by XDC when the programmer attempts to access real (i.e. not virtual) storage.
- The access level that Z/XDC requests is:
  - READ for the initial connection attempt.
  - UPDATE for the initial storage alteration attempt.
- The userid that is passed to security is, of course, the ownerid that security has already associated with the address space in which Z/XDC is running.

If the security system responds with DENIED, then:
- For READ access requests, Z/XDC refuses to let the programmer connect to the targeted address space.
- For UPDATE access requests, Z/XDC aborts the alteration command that the programmer attempted.

If the System responds either with PERMITTED or NOT PROTECTED, then:
- For READ access requests, Z/XDC permits the programmer to connect to the targeted address space.
- For UPDATE access requests, Z/XDC then calls security **again** to see if the programmer is permitted to alter the specific area of storage that he is targeting. This call distinguishes storage according to its location: load module, csect, subpool, freespace, etc. Note that this additional security check is made for all alteration attempts, not just the first. See HELP SECURITY ZAP for more information.

Note, when Z/XDC is running under a clone name other than "XDC", in most instances where the name "XDC" appears as a part of another name (program names, ddnames, VTAM names, etc.), the clone name would be used instead of "XDC". However for security

rules, this substitution does **not** occur! So in RACF protecting systems, the rule strings constructed by Z/XDC remain XDC.FASM.jobname, not clonename.FASM.jobname. (For information about renaming Z/XDC, see HELP XDCCLONES.)

Note that the rule string that Z/XDC builds for controlling Foreign Address Space Mode access to an address space is the **same** string that it builds for controlling access via cdf/XDC. The requested access level, however, for cdf/XDC connections is always UPDATE; whereas, for FASM connections, the initial check made is only for READ access. UPDATE access is not checked until the programmer issues a storage altering command (ZAP, POST, GETMAIN, FREEMAIN, HOOK, etc.). See HELP SECURITY CDF for more information.

Help SEcurity Lostlocks

When Z/XDC is used as an FRR, and the program being debugged had held system locks, then in order for Z/XDC to process, those locks had to be released. (For information about running Z/XDC as an FRR, see HELP DEBUGGING FRR.)

Generally, if the user wishes to resume that program's execution, those locks will have to be reacquired. That is an **extremely dangerous** thing to do!

So before permitting such a resumption to occur, Z/XDC issues a security call to see if the current programmer is permitted to do this. Only programmers expert in the workings of the Operating System should be given permission to perform such resumptions, but only on sandbox systems, **never** on a production system!

Why Losing Locks is Dangerous

The problem with losing and then reacquiring system locks is this:

- Locks exist to protect system integrity.
- In order for them to work correctly, a lock needs to be maintained for the **entirety** of the time that it is needed.
- But when Z/XDC receives control, all locks must be release. Therefore, if some other process was waiting for that lock, then that other process is now allowed to run.
- The problem is, the resource being protected by the lock may be in an undefined (i.e. broken) state because the program running under Z/XDC may not have had time to complete whatever it was doing to or with the resource that had been protected.
- Thus, the other process could now be working with something that is broken, and then break it further.
- Even worse, if the program being debugged needs to resume, it will need to have its locks back, so Z/XDC gets them for it. So the program can then proceed, not realizing that corruption may already have occurred. So it can easily compound the problem.

Because these are **System** locks, problems such as this can lead to utterly random things happening, including (if you're lucky) System crashes.

The Benefit of Debugging Locked Code

On the other hand, if no interference occurs while the locks are lost, then useful debugging can be accomplished within what otherwise would be an exceedingly difficult programming environment to penetrate. So Z/XDC's capability to debug within FRR protected routines can be at least as useful as it is dangerous.

So the FRR debugging capability has been implemented for those who need it, and the XDC.LOSTLOCKS security rule has been implemented for those sites and systems where it cannot be tolerated.

Recommendations

1) - Unlike all of Z/XDC's other security rules, when Z/XDC detects that an XDC.LOSTLOCKS rule has not been defined, it will react as if permission has been **denied!** (For all other rules, Z/XDC reacts as if permission has been granted.) Therefore, do not define an XDC.LOSTLOCKS rule unless you have a specific need to do so.

2) - **Never** define an XDC.LOSTLOCKS rule on a production system! Only allow it to be defined on a sandbox or other system where crashes and frequent IPLs can be tolerated.

3) - Never define a PERMIT to this rule for any but those programmers who are the most experienced both with writing FRR protected code and with the meanings and uses of System Locks, as well as the detailed consequences of losing them.

Z/XDC's Implementation of the XDC.LOSTLOCKS Rule

When Z/XDC receives control and locks had been held, those locks will now be lost, and the user will be warned about that.

The user, for the time being is allowed to continue with the debugging session; however, when he attempts to use a TRACE command or a GO/GOT/GOX command to resume the execution of his routine, Z/XDC calls the Security System to see if the user is allowed to do this.

When Z/XDC makes the security call, it does so as follows:
- The rule string that Z/XDC builds and passes to security is:
  - "XDC.LOSTLOCKS" for RACF protected systems.
  - "LOSTLOCKS" otherwise.
- The access level that Z/XDC requests is READ.
- The userid that is passed to security is, of course, the ownerid that security has already associated with the address space in which Z/XDC is running.

If the security system responds **either** with DENIED or **NOT PROTECTED,** then Z/XDC refuses to allow the user's program to be resumed. Instead, Z/XDC remains in control and solicits additional commands from the user. The only way he is allowed to end

the debugging session is with the END command.

If the System responds with PERMITTED, then:
- Z/XDC restores the user program's retry level environment, including reacquiring all lost locks.
- Z/XDC then passes control back to the user program, allowing it to continue, unaware that locks had been lost and corruption may already have occurred.

Note, when Z/XDC is running under a clone name other than "XDC", in most instances where the name "XDC" appears as a part of another name (program names, ddnames, VTAM names, etc.), the clone name would be used instead of "XDC". However for security rules, this substitution does **not** occur! So in RACF protecting systems, the rule string constructed by Z/XDC remain XDC.LOSTLOCKS, not clonename.LOSTLOCKS. (For information about renaming Z/XDC, see HELP XDCCLONES.)

Help SEcurity Zap

Whenever a programmer issues a Z/XDC command that alters storage, Z/XDC issues a security system call to see if the current programmer is permitted to alter the targeted storage. Z/XDC's security call distinguishes areas of storage according to load module name, csect name, subpool number, free space, and other characteristics. Storage altering commands are ZAP, POST, HOOK, GETMAIN, FREEMAIN, AT, ATX, TRAP, TRACE, LOAD module, and DELETE module.

This security call is made for **all** attempts to alter storage regardless of where that storage is located: in the private area, in common storage, in a data space, in a foreign address space, or in real storage. In the case of alterations to a foreign address space, this security call is made in addition to the "XDC.FASM.---" security calls that are made when the programmer first attempts to access and then to alter a given foreign address space. See HELP SECURITY FASM for more information.

This security call also is made regardless of whether Z/XDC is running authorized or non-authorized.

This security call is **not** made for alterations of the current program's registers (e.g. "ZAP R14=...") and its resume execution address (e.g. "ZAP PSW,...").

The security call that Z/XDC makes has the following characteristics:
- The rule string that Z/XDC builds and passes to security is:
  - "XDC.ZAP.description" for RACF protected systems.
  - "ZAP.description" otherwise.

  where "description" describes the area of storage that the programmer is trying to alter.
- The access level that Z/XDC requests is UPDATE.
- The userid that is passed to security is, of course, the userid that security has already associated with the Home Address Space in which Z/XDC is running.

If the security system responds with DENIED, then Z/XDC aborts the alteration command that the programmer is attempting.

If the System responds either with PERMITTED or NOT PROTECTED, then Z/XDC performs the requested alteration.

Note, when Z/XDC is running under a clone name other than "XDC", in most instances

where the name "XDC" appears as a part of another name (program names, ddnames, VTAM names, etc.), the clone name would be used instead of "XDC". However for security rules, this substitution does **not** occur! So in RACF protecting systems, the rule strings constructed by Z/XDC remain XDC.ZAP.description, not clonename.ZAP.description. (For information about renaming Z/XDC, see HELP XDCCLONES.)

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

DESCRIPTION - A detailed description of the operands that Z/XDC adds to the "XDC.ZAP.---" rule strings that it builds.

OLDRULES - A discussion of issues regarding the changing format of security rules for protecting load modules and program objects.

SUGGESTIONS - Suggestions about how to organize XDC.ZAP rule definitions.

Help SEcurity Zap Description

Whenever a programmer issues a Z/XDC command that may alter storage, Z/XDC builds a rule string to describe the alteration attempt and then calls system security to see if the alteration should be permitted or denied. See HELP SECURITY ZAP for more information.

The rules that Z/XDC builds always start with "XDC.ZAP.---" in RACF protected systems and with "ZAP.---" in ACF2 and Top Secret protected systems. In all systems, the rule strings continue with additional text that more precisely describe the areas of storage that the programmer wants to alter.

Note, when Z/XDC is running under a clone name other than "XDC", in most instances where the name "XDC" appears as a part of another name (program names, ddnames, VTAM names, etc.), the clone name would be used instead of "XDC". However for security rules, this substitution does **not** occur! So in RACF protecting systems, the rule strings constructed by Z/XDC remain XDC.ZAP.description, not clonename.ZAP.description. (For information about renaming Z/XDC, see HELP XDCCLONES.)

For each alteration attempt, Z/XDC calls security either once or twice. The first call is made with regards to the location of the first byte to be altered. The second call is made with regards to the address of the last byte to be altered, but the second call is made (a) only if security's response to the first call was not DENIED, and (b) only if the applicable rule string for the second call is different from the string used for the first call.

Z/XDC analyzes a given alteration address to determine which of the following possible rule strings applies. If more than one rule string applies, then only the first rule string found is used. (Multiple security calls for any one address are not made.) Rule strings are searched for in the order listed here.

XDC.ZAP.REAL
This rule string is used if the target location is anywhere in real storage.

XDC.ZAP.DATASPACE
This rule string is used if the target location is anywhere in any data space.

XDC.ZAP.PRIVATE.---
If an alteration target is located anywhere within a private area of any address space, then Z/XDC builds a rule string that starts with the above text.

Note, for foreign address spaces, Z/XDC has already made "XDC.FASM.---" security calls to obtain general permission for the current programmer to access and alter the Target Address Space. See HELP SECURITY FASM for more information.

XDC.ZAP.COMMON.---
If an alteration target is located anywhere within common storage, then Z/XDC builds a rule string that starts with the above text.

XDC.ZAP.PRIVATE.MODULE.---
XDC.ZAP.COMMON.MODULE.---
If an alteration target is located anywhere within a load module or program object, then Z/XDC builds a rule string that starts with the above text, and contains additional text that identifies the module's name and (possibly) the target csect or other information. "---" will be resolved as follows:

XDC.ZAP.PRIVATE.MODULE.modulename.csect
XDC.ZAP.COMMON.MODULE.modulename.csect
Usage:
- If an alteration target is located within a load module or program object (that is identified as such by z/OS system control blocks: CDEs or LPDEs located on appropriate queues),
- And if the user has previously issued a MAP command for that load module thereby allowing Z/XDC to know where the module's csects are located,
- And if the alteration target is, in fact, located within one of the module's csects,

Then Z/XDC will call security using the rule string shown above. Note, "modulename" will always be the load module's major name, never an alias name.

XDC.ZAP.PRIVATE.modulename.FREEAREA
XDC.ZAP.COMMON.modulename.FREEAREA
Usage:
- If an alteration target is located within a load module or program object,
- And if the user has previously issued a MAP command for that load module,
- And if the alteration target is located in alignment space that sometimes occurs between csects,

Then Z/XDC will call security using the rule string shown above. Note, "modulename" will always be the load module's major name, never an alias name.

XDC.ZAP.PRIVATE.MODULE.modulename.UNMAPPED
XDC.ZAP.COMMON.MODULE.modulename.UNMAPPED
Usage:
- If an alteration target is located within a load module or program object,
- And if the user has **not** previously issued a MAP command for that load module,

Then Z/XDC will call security using the rule string shown above. Note, "modulename" will always be the load module's major name, never an alias name.

XDC.ZAP.PRIVATE.MODULE.modulename
Usage:
- If the programmer is using Z/XDC's "LOAD" or "DELETE" command against a load module or program object,

Then Z/XDC will call security using the rule string shown above. Note, the text "PRIVATE" is used in this rule even if the module being LOAD'd is actually located in common storage. This is because of the following:
- Z/XDC's LOAD command can only:
  - Read modules into the Home Address Space's private area, or
  - Create a connection from the Home Address Space to a load module that already resides in common storage.

  The LOAD command cannot actually alter common storage.
- Similarly, the DELETE command can only remove a load module from the private area or disconnect the Home Address Space from a common area module. It cannot actually alter common storage.
- For DELETE commands, "modulename" will always be the load module's major name.
- However, the same is **not** true for LOAD commands: "modulename" will be the name that the programmer actually uses on the LOAD command, and that **could** be an alias name.

NOTICE: Prior to release z1.7 of z/XDC, the rule strings for load modules and program objects did not contain the keyword ".MODULE". Example, for zaps against IEFBR14:
- Current Rule: XDC.ZAP.COMMON.MODULE.IEFBR14.UNMAPPED
- Old Rule: XDC.ZAP.COMMON.IEFBR14.UNMAPPED

Z/XDC checks for the current rule first, but if an applicable rule is not found, then it checks again for an old rule.

For the time being, if Z/XDC finds an old rule and it does **not** find a new rule, then it will obey that old rule. However, support for the old rules will be dropped in a future release, so it is important that Security Officers replace their old rules with new ones. For more information, see HELP MESSAGES DBC981.

This change in rule formats is, admittedly, inconvenient; however, there is a pretty good reason for doing so. The now format permits the Security Officer to create a global rule **(XDC.ZAP.COMMON.MODULE.*)** for defining a default protection for all load modules located in shared storage. Such a rule was not possible to create under the the old format.

XDC.ZAP.PRIVATE.SUBPOOL.---
XDC.ZAP.COMMON.SUBPOOL.---
If Z/XDC cannot determine that an alteration target is located within a load module (or program object), then it will try to determine what storage subpool it is located in. "---" will be resolved as follows:

XDC.ZAP.PRIVATE.SUBPOOL.UNKNOWN
- If the alteration target is in the private area of a foreign address space,
- And if it is not located within any load module,

Then Z/XDC will call security using the rule string shown above. Z/XDC does not, at this time, have the technology necessary to identify subpools in a foreign address space.

XDC.ZAP.PRIVATE.SUBPOOL.number
XDC.ZAP.COMMON.SUBPOOL.number
- If Z/XDC can determine that the alteration target is within a subpool,
- Or if the programmer is using Z/XDC's "GETMAIN" command to obtain storage in

a subpool,
Then "number" is the decimal number of that subpool. Please note the following:
- "Number" is always 3-digits wide. Leading zeros are used when necessary.
- For GETMAINs, if an authorized program requests subpool 250, z/OS will actually supply subpool zero instead. Accordingly, if the user specifies subpool 250 on Z/XDC's GETMAIN command, the rule string that Z/XDC builds will actually have "000" for "number".

XDC.ZAP.PRIVATE.SUBPOOL.SYSRGN
If the alteration target is within the first four pages of the private area of the user program's Home Address Space, then it is not within any subpool. Instead, it is in an area of storage known the "System Region". This is private area storage, but it is not available for use by the user program. Instead it is reserved for use by the address space's Region Control Task and the System Dump Task.

XDC.ZAP.COMMON.SUBPOOL.LOWCORE
If the alteration target is within:
- The first page of virtual storage (for OS/390 systems)
- The first **two** pages of virtual storage (for z/OS systems)

Then it is not within any subpool. Instead, it is in an area of storage known as the "Prefixed Storage Area" (PSA). This is special use common area storage.

XDC.ZAP.COMMON.SUBPOOL.PLPADIR
If the alteration target is within the area of storage that is set aside for the Pageable Link Pack Area, but it is not within any load module, then it most likely is in the PLPA directory; although, it could also be located within inter-module alignment space. In either case, security is called with the above rule string.

XDC.ZAP.PRIVATE.SUBPOOL.FREEAREA
XDC.ZAP.COMMON.SUBPOOL.FREEAREA
If the alteration target cannot be determined to be in one of the areas described above, then this rule string "of last resort" is used.

Help SEcurity Zap Oldrules

Currently, Z/XDC is in the process of converting its support for the rules that protect the zapping of load modules and program objects. The old naming format for such rules was:

```
XDC.ZAP.COMMON.modulename.---
XDC.ZAP.PRIVATE.modulename.---
```

The new format is:

```
XDC.ZAP.COMMON.MODULE.modulename.---
XDC.ZAP.PRIVATE.MODULE.modulename.---
```

In other words, the keyword "MODULE" has been added to the rule string.

This change is being made to make it easier to protect modules generically without, at the same time, affecting the protection of data areas.

During a conversion period, Z/XDC will permit the protection of modules using either the old or new format rule strings by checking **both** formats and analyzing the results to see which of the two rulings to follow.

Unfortunately, there are circumstances under which the two rulings will contradict each other, and no additional information will be available for deciding which ruling to follow. When this kind of ambiguity arises (and **only when** this kind of ambiguity arises), Z/XDC then checks to see whether or not the current user has at least READ access to a "referee" rule named **XDC.REFEREE.ZAPMODULERULES.** Z/XDC then proceeds as follows:

- **Permitted:**
  The user's zap, trace, or breakpoint attempt is either permitted or denied according the **new** format rules.

- **Denied:**
- Or **Not Protected:** (i.e. the XDC.REFEREE.ZAPMODULERULES rule does not exist)
  The user's zap, trace, or breakpoint attempt is either permitted or denied according the **old** format rules. In addition, message DBC953W is issued to alert the user that a new format security rule may have been ignored.

Why and When the XDC.REFEREE.ZAPMODULERULES Referee Rule is Needed

Because Security Officers can create generic rules, it is possible for ambiguities to arise when one of Z/XDC's checks is responded to by a relatively specific rule, and the other check is responded to by a relatively non-specific rule. Example:

RULE (for load module zaps)	**RULING**
XDC.ZAP.COMMON.MODULE.pgmname.*	PERMIT
XDC.ZAP.COMMON.*	DENY

This is a reasonable combination of rules for a Security Officer to create, but when Z/XDC asks security for a ruling using the new format (XDC.ZAP.COMMON.MODULE.pgmname.csectname), security will find the first rule and respond with "permit". But when Z/XDC asks again using the old format (XDC.ZAP.COMMON.pgmname.csectname), security will find the second rule and respond with "deny".

Here's the opposite example:

RULE (for load module zaps)	**RULING**
XDC.ZAP.COMMON.pgmname.*	PERMIT
XDC.ZAP.COMMON.*	DENY

In this case, when Z/XDC asks security for a ruling using the new format (XDC.ZAP.COMMON.MODULE.pgmname.csectname), security will find the second rule and respond with "deny". But when Z/XDC asks again using the old format (XDC.ZAP.COMMON.pgmname.csectname), security will find the first rule and respond with "permit".

When Z/XDC receives such contradictory rulings, it needs to have additional information in order to decide which ruling to follow and which ruling to ignore. Specifically, Z/XDC needs to see the actual rules that gave rise to the contradictory rulings.

Now, sometimes Z/XDC can see the rule that security uses to make a ruling, and when it can, it then does decide for itself which ruling is the appropriate one to accept. But when the underlying rule is not available for Z/XDC to examine, another way is needed to permit Z/XDC to make a decision. That other way is for Z/XDC to check a "referee" rule named **XDC.REFEREE.ZAPMODULERULES** and (as described above) choose to follow the new or old ruling according to whether or not the user has READ

access to the referee rule.

Corrective Action:

When the referee rule is not properly set up when needed, Z/XDC will alert the user to the problem by issuing message DBC953W. When this occurs, you need to ask your Data Center's Security Officer to take the following actions:

- Create new format security rules for controlling your accesses to the load modules (or program objects) that you are trying to zap or trace or breakpoint into.

- Unless an older release or version of XDC is still in use (i.e. z/XDC z1.6 or older), delete the old format rule that had previously been controlling such accesses.

- If it hasn't been done already, then create a rule named "XDC.REFEREE.ZAPMODULERULES".

- Give you READ access to the "XDC.REFEREE.ZAPMODULERULES" rule.

Help SEcurity Zap Suggestions

The "XDC.ZAP" security rules are rather complex, so it seems appropriate to offer some suggestions for their reasonable usage. Of course, your shop's Security Officer needs to make his own determinations of what's best locally.

- Generally, there should be no problem with allowing a user to zap any accessible location in his own private area. Accordingly, even though Z/XDC's "XDC.ZAP.PRIVATE.---" rules allow rather detailed control of private area zaps, I would generally recommend not using them. Instead, I would suggest defining an "XDC.ZAP.PRIVATE.*" rule with a default access of UPDATE to permit all users to have zap access to their own private areas. A rule such as this is especially necessary, if you also have an "XDC.ZAP.*" rule (which is more general than the "XDC.ZAP.PRIVATE.*" rule) that has a default access of NONE. RACF Example:

```
RDEFINE FACILITY XDC.ZAP.PRIVATE.* UACC(UPDATE)
RDEFINE FACILITY XDC.ZAP.*         UACC(NONE)
RDEFINE FACILITY XDC.FASM.*        UACC(NONE)
```

- The "XDC.ZAP.PRIVATE.*" rule suggested above would allow a user, running APF authorized, also to have unrestrained access to the private areas of foreign address spaces, BUT ONLY IF the user is first given access to the address space via the XDC.FASM.--- rules. If a user is denied access to a foreign address space via an XDC.FASM.--- rule, then the XDC.ZAP.PRIVATE.* rule has no effect. In other words, the primary method for protecting foreign address spaces should be through the XDC.FASM.--- rules, not the XDC.ZAP.--- rules.

- The protection of specific load modules should be considered only for load modules located in common storage. You might start with a blanket rule that denies zap access to all common storage (load modules, subpools, everything), and then grant exceptions as needed. RACF Example:

```
RDEFINE FACILITY XDC.ZAP.COMMON.* UACC(NONE)
RDEFINE FACILITY XDC.AUTH         UACC(NONE)
```

Remember, a user won't be able to zap common storage in the first place unless he is running APF authorized, and Z/XDC's primary gatekeeper for that power is the "XDC.AUTH" rule.

- Note, older customers who used to use the old "XDC.GZAP" rule to control zaps against common storage can achieve the identical control by creating a rule named **XDC.ZAP.COMMON.*** and assigning to it the same permits that had been assigned to the XDC.GZAP rule. XDC.ZAP.COMMON.* can, of course, coexist with other XDC.ZAP.--- rules and will be checked only for those shared storage zaps not covered by more specific rules.

- When forming security rules to control alterations to load modules, keep in mind that whether or not Z/XDC has a csect map available is under the **programmer's** control. Therefore, rules that prevent zaps to specific csects will not be effective unless there is also a rule that prevents zaps to the load module when it is unmapped.

  General Example: Suppose that a load module named BIGPROJ has two csects. One is named SUZYS, and the other is named JOES. Then in order to allow Joe to zap his own csect but not Suzy's (and vise-versa), you would have to create the following security rules:

```
XDC.ZAP.COMMON.BIGPROJ          (controls LOADs and DELETEs)
    Suzy - permit
    Joe  - permit
XDC.ZAP.COMMON.BIGPROJ.UNMAPPED (controls zaps prior to mapping)
    Suzy - deny
    Joe  - deny
XDC.ZAP.COMMON.BIGPROJ.SUZYS    (controls zaps to individual csects)
    Suzy - permit
    Joe  - deny
XDC.ZAP.COMMON.BIGPROJ.JOES     (controls zaps to individual csects)
    Suzy - deny
    Joe  - permit
XDC.ZAP.COMMON.BIGPROJ.*        (controls zaps to all other csects)
    Suzy - deny
    Joe  - deny
```

  Neither programmer would be allowed to zap any part of BIGPROJ until they mapped it. Once mapped, csect-specific permits and denials then become effective. Also note, in this example both programmers would be allowed to issue Z/XDC's LOAD and DELETE commands to bring copies of BIGPROJ in and out of storage as they saw fit.

- When forming security rules to control alterations to the System Nucleus, keep in mind that the Nucleus' name is usually IEANUC01, but only usually. The last digit can, depending upon the IPL that was last performed, be changed from "1" to any other character. Therefore, rules for the Nucleus should specify a wildcard character for this position.

- Any location that has a virtual address also, upon reference, will always have a real address as well. If a programmer is blocked from zapping a location via its virtual address, he might (if he's running APF authorized) attempt to perform the same zap via its real address instead. For this reason (and many others), you may want to use the "XDC.ZAP.REAL" rule to prevent users from zapping real storage.
  RACF Example:

```
    RDEFINE FACILITY XDC.ZAP.REAL     UACC(NONE)
```

Help SEcurity Gzap

The XDC.GZAP rule is an obsolete security rule that used to be used to prohibit selected Z/XDC users from zapping any of shared storage. However, the introduction, in 1995, of the **XDC.ZAP.---** rules made the XDC.GZAP rule obsolete, and Z/XDC documentation was changed at that time to urge customers to replace their XDC.GZAP rule with suitable XDC.ZAP.--- rules. Nonetheless, support for the XDC.GZAP rule was maintained for the convenience of the Customers.

Beginning with z/XDC z1.2 (released in October 2003), if XDC found itself making a security decision based upon the XDC.GZAP rule it would act upon that decision, but it would also issue a message (DBC980W) to the user warning him that support for the XDC.GZAP rule was about to be dropped, and his Data Center's Security Officer needed to replace the XDC.GZAP rule with XDC.ZAP.--- rules in order to avoid future problems.

Well, now support for XDC.GZAP has, indeed, been dropped, and now customers who have not updated their security rules are having problems. Specifically:

- If Z/XDC has **not** found an XDC.ZAP.--- rule that covers a particular common storage zap attempt,

- And if Z/XDC **has** found an XDC.GZAP rule,

- **Then** Z/XDC will unconditionally prohibit the zap attempt, and it will issue message DBC980E to explain why.

(This action is preferable to pretending that the XDC.GZAP rule did not exist at all, because that would create an unacceptable security exposure.)

To achieve at least the same functionality intended by the XDC.GZAP rule, your security officer must define a rule named **XDC.ZAP.COMMON.*** and assign to it the same permits that had been assigned to the XDC.GZAP rule.

For more information, see HELP SECURITY ZAP.

Help SEcurity ACif

Normally, when Z/XDC calls system security, it does so by calling the System's "SAF" interface. ("SAF" stands for System Authorization Facility" and is always present in z/OS regardless of whether or not a security system is actually installed.) However, Z/XDC also has support for a customer written exit which, if present, Z/XDC will call instead of the SAF interface. This exit is called the "ACIF" exit. ("ACIF" stands for "Access Control Interface".) An ACIF exit may itself call SAF, or it may simply make the security decision itself and return to Z/XDC without calling SAF.

Z/XDC's support of the ACIF exit is designed so that a data center's staff can prevent unauthorized users from installing their own ACIF exits. In particular, when looking for an ACIF exit, Z/XDC requires that the exit be located in common storage. It refuses to use any copy not located in common storage.

For more information, see HELP EXITS ACIF.

Help SEcurity CAche

For performance reasons, whenever Z/XDC makes a security call, it records the result of that call into a local cache. Then to save time, subsequent security queries can be resolved against the cache instead of by the system.

Having this cache is needed because in some cases, such as during conditional tracing, Z/XDC may have to call security hundreds or even thousands of times per second! This is because at each step in a trace, temporary breakpoints have to be set and then removed, and each such event is a storage modification that needs to be permitted by a security call. So anything that can be done to streamline this process can have substantial performance benefits, and Z/XDC's local cache does streamline the process.

However, having a local cache creates a security exposure, since if (subsequent to the creation of a cache entry) a security rule is changed, then Z/XDC will not see that change. So to minimize this exposure, all cache entries are given only a two minute lifetime. When that expires, then the cache entry is discarded and the next security query that would use it, instead is sent, to system security again.

Generally, because the cache entry lives are so short, there isn't much needed in the way of management tools for it; however, the following Z/XDC commands are of interest.

- **LIST SECURITY**
  This command produces a report that has a line showing how much time remains in the life of the youngest cache entry. (That time will never be longer than two minutes.)

- **SET SECURITY FLUSHCACHE**
  If, for some reason, you want to force the immediate expiration of all cache entries maintained by your debugging session, this command will do the trick.

Help SEcurity Trace

- Using the **SET SECURITY TRACE** command, you can activate a trace of all calls made by Z/XDC to its system security interface. For each such call, a multi-line report is displayed. This report is displayed at the user's terminal along with the output of all his other Z/XDC commands.

  Each trace report may (as appropriate) include the following information:

  ZAP CALLED BY:
  (For commands that alter storage): this shows the name of Z/XDC's internal routine that is calling the security interface.

  TARGET ADDRESS:
  (For commands that alter storage): this shows the virtual address that Z/XDC

(pursuant to a user's command) wants to alter.

SECURITY CALLED:
This shows which of the major security systems is installed, RACF, ACF2, or TSSF (CA-Top Secret). It also shows whether Z/XDC is making the call directly or via it's service SVC.

Alternatively, this report line might show that the ruling is being made from an internal Z/XDC cache entry, in which case it will also show how much time remains before that cache entry expires.

RESOURCE NAME:
This shows the exact resource name that Z/XDC has constructed to represent the resource that is being protected by the security call.

GOVERNING RULE:
If the appropriate feedback is available from system security, then this shows the exact security rule (if any) that security used to make its ruling.

USER NAME:
This shows the ownerid in whose name Z/XDC is making its security calls. When debugging TSO-running programs, this will, of course, match the current TSO userid. When debugging batch jobs, this will be the job's security assigned ownerid (if any). (In all cases, Z/XDC obtains this information from the ACEEUSER field of the ACEE control block.)

RULING:
This shows the ruling that Z/XDC obtained from system security (or from its internal cache). It might be any of the following:
- ACCESS PERMITTED
- ACCESS DENIED
- NOT PROTECTED

FINAL DECISION:
This shows Z/XDC's final access decision. Generally, this will match the security system's ruling; however, in some cases (such as when Z/XDC needs to determine and resolve rulings based upon both old format and new format rules), Z/XDC might chose not to follow a particular ruling but instead evaluate multiple rulings in order to make its decision.

Help Tachyon

Tachyon Software makes an Assembler named **z/Assembler.** This is a great(!) Assembler that's worth talking about.

First, z/Assembler is a "cross Assembler". This means that it executes on one kind of computer but produces object code for execution on a different kind of computer. In this case, z/Assembler runs on PCs (DOS, Windows, OS/2, or UNIX) but produces object

code that is Binder ready for the mainframe. All you have to do is upload the z/Assembler output to your mainframe, run it through the Binder, then execute it. This means that you can actually offload the assembly process from your mainframe to your PC giving you greater control over when and where you do your work and relieving the mainframe of some of its overload.

Second, z/Assembler is an essentially complete and well supported PC implementation of IBM's High Level Assembler that in many cases actually runs **faster** than IBM's version.

Third, Tachyon Software has implemented several extensions to SYM record output (see HELP MAPS SYM) specifically for Z/XDC's benefit that help improve Z/XDC's usability during debugging. The extensions are:

LONG NAMES
Starting with Assembler H and continuing with the High Level Assembler, IBM has permitted programmers to define statement labels and field names that can be up to 63 characters long. Unfortunately, IBM did not update the Assemblers' SYM record output format (Z/XDC's primary information source for labels) to support the longer names. This is why Z/XDC was unable to display long names in its csect and dsect maps. (SYM data are object file records that an Assembler produces when "PARM=TEST" is specified.)

In the Fall of 1996, Tachyon's Dave Bond and ColeSoft's Dave Cole developed an IBM compatible extension to the SYM record format so that the Tachyon Cross Assembler could communicate long names to Z/XDC. To enable this support, all you have to do is specify the SYMNAME(XDC) and TEST(XDC) Assembler options. When you do this, any long names that have been defined in your program can be used by Z/XDC for displays and for address expressions. See the following for more information:

HELP ADDRESSING PARSERS ASM LONGNAMES
HELP MAPS SYM

EQU SYMBOLS
Way way back in the dark ages, IBM's VS Assembler (IFOX00, remember him?) would include in its SYM table, entries for relocatable equates (EQU statements). For reasons which I can only guess at, this support was dropped by IBM starting with Assembler H (IEV90) and (of course) continuing with today's High Level Assembler (ASMA90). The consequence of this action is that all labels defined by EQU statements stopped being visible in csect and dsect maps used by Z/XDC. In particular, all labels defined by EQU * statements were lost, and so a lot of Z/XDC's customers had to change their "EQU *" statements to "DS 0X".

Anyway, as of the fall of 1996, Dave Bond has restored this support to his Assembler. So now if you specify the TEST(XDC) Assembler option, the SYM data produced by the Assembler will include entries for all relocatable equates, and they will be usable in Z/XDC's displays and commands. For more information, see HELP MAPS SYM.

SYMDEL/SYMUNDEL/SYMNODEL SUPPORT
Because of certain problems that can arise when linkediting multiple assemblies into a single load module, it is possible that an excessive amount of redundant and unneeded assembly time symbols can accumulate in the load module when the linkedit is done with PARM=TEST specified. (See HELP MAPS SYM EXCESSSYMDATA for details.) To deal with this problem, I wrote a utility named XDCSYMED that can be used to remove unwanted symbols from an Assembler's object file output.

When the Tachyon Cross Assembler is used, XDCSYMED is not needed. The Tachyon Assembler will look for the SYMDEL, SYMNODEL, and SYMUNDEL symbols and perform the very same symbol deletions that XDCSYMED performs. It also will perform all the other editing and cleanup functions that XDCSYMED does. All you have to do is specify the TEST(XDC) Assembler option when you run the Tachyon Cross Assembler. For more information, see HELP MAPS SYM EXCESSSYMDATA TACHYON.

ADATA SUPPORT
Starting with the Summer of 2000, both Z/XDC and X390 (z/Assembler's prior version) came out with new product releases that for the first time provided support for producing (via X390) and using (via Z/XDC) ADATA. z/Assembler now supports the generation of High Level Assembler compatible ADATA from its assemblies, and Z/XDC supports the construction of csect and dsect maps from ADATA found either within a program object, or within a GOFF formatted object file, or in a raw SYSADATA file. (See HELP MAPS ADATA for more information.)

To use ADATA produced by z/Assembler, just upload it (binary mode) to the mainframe into any sequential dataset or PDS(E) member. The RECFM and LRECL of the dataset can be anything. This is because Z/XDC reads ADATA as a streamed file, i.e. without regard to file record boundaries.

Tachyon Software can be reached as follows:

```
FAX:     303-840-2564
VOICE:   303-840-2487
EMAIL:   dbond@tachyonsoft.com
WEBPAGE: http://www.tachyonsoft.com/tachyon

SNAIL    Tachyon Software
MAIL:    12794 N. Sierra Circle
         Parker CO, 80134-8719
```

Help USERInterfaces

Z/XDC supports many different methods for interactive communication with its user, depending upon:

- Whether Z/XDC is being used in a TSO address space or in a background address space.
- Whether or not Z/XDC's Fullscreen Support is turned on.
- Whether or not Z/XDC is running within ISPF.
- Whether or not the user has chosen to use cdf/XDC for interactive debugging of background jobs and system tasks.

Subtopics Index

Specifically, the following interactive communication modes are available. For detailed descriptions, these topics type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

FULLSCREEN - The Z/XDC's Fullscreen Support provides two interchangeable but distinct fullscreen display interfaces for use when debugging programs

running in TSO. One uses fullscreen TPUTs while the other uses ISPF Display Services.

- CDF - cdf/XDC is a component of Z/XDC that provides an interactive interface to debugging sessions running in background jobs or system tasks. cdf/XDC can generate its terminal displays using either fullscreen TPUTs or ISPF service calls.

- LINEMODE - If Z/XDC's Fullscreen Support is turned off, then Z/XDC can use linemode TPUTs to send its displays to the terminal.

- WTOWTOR - If cdf/XDC is not available, then when Z/XDC is being used to debug a background program, then it can use WTOs and WTORs for interactive communication to a user at a System Operator's console.

- UCI - Finally, Z/XDC also supports interactive communications to a user written exit routine called a "User Communications Interface" routine or "UCI" for short.

Each of these several communications modes has its advantages and disadvantages:

- The two linemode interfaces (foreground TPUT/TGETs and background WTO/WTORs) are built into Z/XDC's base code, so they are always available for use at all times in all environments.

- All of the various interfaces provided by the Z/XDC Fullscreen Support are far more powerful and versatile than the linemode interfaces; however, there occasionally are circumstances when they cannot be used (such as when debugging a VTAM exit).

Regardless of which interface you may generally prefer, another interface may occasionally be preferable in an given specific situation. Switching, however, among the available interfaces can generally be done at will just by typing a command or two.

Help USERInterfaces Fullscreen

Z/XDC's Fullscreen Support provides two interchangeable but distinct fullscreen display interfaces for use when debugging programs running in TSO. One uses fullscreen TPUTs to generate displays at the terminal while the other uses ISPF Display Services. The two interfaces are called the **TPUT interface** and the **ISPF interface,** respectively.

Either fullscreen interface can be used regardless of the nature of the program being debugged. In other words, either fullscreen interface can be used to debug both ISPF dialogs as well as other programs. The ISPF interface, however, can be used only when an ISPF environment exists and is available for use by Z/XDC. The TPUT interface, on the other hand, can be used in all TSO address space environments.

The two fullscreen interfaces are nearly identical in appearance and capabilities. In fact, when the programming environment supports the use of the ISPF interface,

the user can instantly switch back and forth between the two interfaces just by issuing a **SET ISPF** or **SET TPUT** command.

The ISPF interface permits use of the **SPLIT, SWAP** and ISPF **=jump** commands (e.g. **=3.3).** However, the ISPF interface cannot be used if Z/XDC is running outside of ISPF or if Z/XDC is running authorized.

The TPUT interface, on the other hand, does not support the **SPLIT, SWAP** and ISPF's **=jump** commands. However, this interface can be used regardless of whether or not ISPF is available and regardless of whether or not Z/XDC is running authorized.

Both fullscreen interfaces provide a wide range of display and windowing options as well as PF key support, profile support, and session scrolling and logging. The fullscreen interfaces also provide a wide variety of input fields and command entry methods such as item selection, data overtyping, Point-and-Shoot commands, Built-in Help with crosslinks, multiple primary command lines, etc. See **HELP FULLSCREEN** for detailed information.

Both fullscreen interfaces can be used at the terminal end of a background job debugging session via cdf/XDC regardless of whether the background job is authorized or non-authorized. This is because the terminal end of the session runs non-authorized under ISPF in TSO. See **HELP XDCSRVER CDF** for more information.

Fullscreen interface selection is controlled by the following commands.

SET TFS OFF
This command turns off the fullscreen interface (either one) in favor of a linemode interface. If the Z/XDC debugging session is running in a TSO address space, then it will fall back to using linemode TPUT/TGETs to communicate with the user. On the other hand, if the debugging session is running in a background job or system task, then Z/XDC suspends communicating via cdf/XDC and starts using WTO/WTORs instead. See **HELP USERINTERFACES LINEMODE** for more information.

SET TFS ON
This command turns Z/XDC's Fullscreen Support back on. Thus Z/XDC discontinues using linemode TPUTs and resumes using one of the fullscreen interfaces (whichever one it was using when the **SET TFS OFF** command was issued).

SET TPUT
This command causes Z/XDC to discontinue using its ISPF interface and start using its TPUT interface instead. The following will occur:
- The screen's title line will change slightly.
- ISPF's **SPIT, SWAP** and **=jump** commands will become UNavailable for use.

SET TPUT cannot be issued when **SET TFS OFF** is in effect.

SET ISPF
If the current programming environment supports it, then this command causes Z/XDC to discontinue using its TPUT interface and start using its ISPF interface instead. The following will occur:
- The screen's title line will change slightly.
- The **SPLIT, SWAP** and ISPF's **=jump** commands will become available for use.

SET ISPF cannot be issued when **SET TFS OFF** is in effect. Also, it will be ignored unless it is issued from within an ISPF programming environment. This means that Z/XDC must be debugging a program that is running under ISPF, and that program must be running non-authorized. It is irrelevant whether or not the program being debugged is itself an ISPF dialog.

The Z/XDC controlled linemode interfaces display messages as they are generated, whereas the fullscreen interfaces buffer all messages from Z/XDC until Z/XDC makes a "get" request to read the next command from the user. Consequently, for those few Z/XDC commands that are long running, progress messages will not appear at the terminal until the commands complete. Note the following specific cases:

MAP and **DMAP**
When these commands **begin** to read a symbol table from disk, they issue a message saying so, but because of the fullscreen buffering delay, this message is not actually displayed until the reading process completes. Thus, the tense of the message implies that the reading of the map is in progress while in fact by the time the message is displayed, the command has already completed and the keyboard has been unlocked to accept new input.

FIND
This command does not issue any progress messages, but it can take a long time to scan storage, and so the keyboard can remained locked for a long time. If you wish, you can use an attention signal to interrupt the storage scan, and the point of interruption will be reported to you.

READ
This command can take a long time, especially when the script being read includes several **MAP** or **DMAP** commands. However:

- If **Read Echoing** is in effect (see **HELP COMMANDS SET READ),**
- And if the program that you are debugging is running within TSO (Read Echoing is not effective when debugging via cdf/XDC),

Then all messages generated by Z/XDC will be displayed via linemode TPUTs **as they are generated,** so you will be able to monitor the progress of the script.

Copies of all messages still will also be buffered by Z/XDC's Fullscreen Support for fullscreen mode display when the **READ** command completes.

Help USERInterfaces Cdf

cdf/XDC is a very easy to use communications device that allows a user at either a TSO terminal or an idle VTAM terminal to connect to and engage in a fullscreen interactive debugging session with a Z/XDC that is running in a background job or system task.

In order to use cdf/XDC, a background job must first ABEND or reach a breakpoint such that Z/XDC receives control in the background address space. Z/XDC then decides how it is going to communicate with a user. Its default action is to use cdf/XDC, but under certain circumstances it will try to use its WTO/WTOR interface instead. See **HELP XDCSRVER CDF COMMUNICATIONS** for more information.

If Z/XDC does use cdf/XDC, then it connects to the Cross Domain Facility, issues a WTOR (DBC640Q ... AWAITING PROGRAMMER SIGNON ...), and waits for a programmer to sign on to the debugging session. At this point, a user can, subject to various security checks, also connect to cdf/XDC and select the job for debugging.

A use may connect to cdf/XDC either from an idle VTAM terminal or from a TSO

session:

- From an idle VTAM terminal, type one or the other of the following commands:
  LOGON APPLID=xxx
  LOGON APPLID(xxx)
  where **xxx** matches Z/XDC's current name (usually **XDC).** You will then be presented with a logon screen where you must give your TSO userid and password. After that, you will be presented with a selection list of jobs awaiting debugging.

- From an ISPF session, display Z/XDC's Startup Panel (usually **=D,** but you may have to ask your Systems Programmer where in ISPF he installed that panel.) Then select option **3.** The logon to cdf/XDC will be automatic, and you will be taken directly to a job selection list.

For more information about cdf/XDC, see **HELP XDCSRVER CDF.**

Help USERInterfaces Linemode

When running in a TSO address space, if Z/XDC's Fullscreen Support is turned off (by a **SET TFS OFF** command, for example), then communication is performed via linemode TPUTs and TGETs directed towards the user's terminal.

This is the only mode of communication to a TSO terminal that **displays messages in real time,** as they are generated. The other TSO terminal communicating modes buffer all Z/XDC generated messages and do not display them until just before Z/XDC issues a **GET** request to read the next command from the terminal. This delay is noticeable for such long running commands as:

DMAP
FIND
MAP
READ

Z/XDC's Fullscreen Support can be turned on and off via the **SET TFS** command. See **HELP COMMANDS SET TFS** for more information.

When the Z/XDC controlled linemode interfaces are in use, all commands that are supported by the Z/XDC's Fullscreen Support are invalid. Exception: The **SET TFS ON** command to turn the Fullscreen Support back on. In particular, a **SET TFS ON** command must be issued before you can use the

- **SET ISPF**
- or **SET TPUT**

commands to select a particular fullscreen interface.

Help USERInterfaces Wtowtor

When Z/XDC receives control (due to an ABEND or breakpoint) in a background job or system task, it must decide how it is going to communicate with a user. For this

initial decision, Z/XDC's default action is to use cdf/XDC, but the user can, if he wants, override this default and force Z/XDC to use its WTO/WTOR interface instead. To do so:

- **(1)** Add a **//xxxWTOR DD DUMMY** DD-card to the job's JCL.

- **(2)** Ensure that a **//xxxCDF DD DUMMY** DD-card is **not** present. (See **HELP XDCSRVER CDF COMMUNICATIONS** for more information.)

If **//xxxWTOR DD DUMMY** and **//xxxCDF DD DUMMY** are **both** present, WTO/WTOR becomes the **fallback** interface:
- cdf/XDC is tried first.
- The WTO/WTOR interface is tried only if cdf/XDC fails for some reason.

- (Note, one reason would be Operations replying **C** to cdf/XDC's **DBC640Q** query.)

Note, Z/XDC's WTO/WTOR interface is not intended for use by a data center's general user community. Normally, the operator consoles need to be reserved for use by System Operators for the purpose of running the System, and the large numbers of messages that can be generated by Z/XDC can be disruptive. Nonetheless, there are rare circumstances that might occur in certain systems program development situations wherein the other communications path offered by cdf/XDC cannot be used.

If Z/XDC is directed to use WTO/WTORs, it first issues a WTOR **(DBC829Q)** asking the System Operators for permission to proceed with a debugging session. If an Operator replies **NO,** then Z/XDC terminates and percolates the ABEND to the next older ESTAE, ESTAI, or ESTAEX routine.

On the other hand, if the Operator replies **YES,** then Z/XDC commences to send its displays to the operator consoles and to accept commands via replies to its WTORs.

Once Z/XDC begins using consoles for conducting the debugging session, the routing of messages is automatic. Whenever Z/XDC receives a session command, it will identify the console from which that command was issued, and it will route all response messages back to that same console.

If you eventually want to switch the debugging session away from operator consoles and over to another mechanism (cdf/XDC, for example), you can try issuing a **SET CONSOLES OFF** command.

If another communications path is available, then Z/XDC will attempt to use it. Otherwise, the attempted switch will be ignore, and Z/XDC will continue using operator consoles. For more information, see **HELP COMMANDS SET CONSOLE.**

Note, when **dump/XDC** is active, **//xxxWTOR DD DUMMY** is ignored.

Help USERInterfaces Uci

If the user writes a "User Communication Interface" exit routine ("UCI"), then it is used for communication with the user. See HELP EXITS UCI for more information. Also, the SRCONLHG member of the XDCASM library contains a sample program that uses the UCI interface. (The library's factory default name is hlq.XDCV31.XDCASM. Ask your Systems Programmer for its actual name at your Data Center.)

For more information, see **HELP EXITS UCI.**

Help UTilities

Included with the Z/XDC product are several support **utilities** that are of use in preparing for or assisting in the use of Z/XDC itself.

The utilities are separate programs that you would typically run in a batch job to accomplish some particular purpose related to the debugging process.

Specifically, the following utility programs are available. For details, type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

XDCADATA - Edits a file containing ADATA to remove redundant or otherwise unneeded ADATA records. File size reductions in the range of 80 to 90% and more can be achieved.

XDCCALL - An optional utility that creates a debugging session for your **non-authorized** program. It performs numerous required and optional setup services for the debugging session.

XDCCALLA - Same as XDCCALL, but for **authorized** programs.

XDCCMD - Identical to XDCCALL, but for debugging non-authorized **TSO Commands.** The only difference is that upon entry, R1 points to a **CPPL** instead of to a PARM field.

XDCCMDA - Identical to XDCCMD, but for **authorized** TSO Commands.

XDCMMEF - Creates a Module Mapping Extraction File **(MMEF)** containing extracted and compressed mapping data for any/all load modules in one or more load libraries.

XDCSYMED - Edits an object file (a //SYSLIN file) that contains SYM data. It removes redundant or otherwise unneeded SYM data entries, resulting in significant reductions in a load module’s size on DASD. (Does not affect its size in storage.)

Help UTilities XDCAdata

The XDCADATA utility can be used to remove from a file of ADATA all unneeded and most redundant ADATA records. When properly used, **file size reductions of 80 to 90%** or more can be achieved!

ADATA is created by the high Level Assembler and is written either to a side file (via //SYSADATA) or to GOFF output (via //SYSLIN).

When building a product from multiple assemblies, there typically will be a large amount of ADATA output (CVT maps and the like) that is duplicated in each assembly. The XDCADATA utility, working in conjunction with certain otherwise innocuous source code changes, can detect this redundancy and eliminate it. For more information, see **HELP MAPS ADATA EXCESSADATA.**

To use XDCADATA, you need to:

- Add **ADATA** to the assembly step's **PARM** field to cause the Assembler to write ADATA to a DD named **//SYSADATA.**

- Add **//SYSADATA DD** to send the output ADATA to a temporary VB-27994 file to be passed to the ADATA editing step. Example:

```
//SYSADATA  DD DSN=&&ADATA,
//             DCB=(RECFM=VB,LRECL=27994,BLKSIZE=0),
//             UNIT=SYSDA,DISP=(,PASS),
//             SPACE=(TRK,(5000,1000),RLSE)
```

- Add a **2nd step** to your assembly proc with JCL such as the following:

```
//AFILTER EXEC PGM=XDCADATA,REGION=0M
//XDCPRINT  DD SYSOUT=*
//SYSADATA  DD DISP=(OLD,DELETE),DSN=&&ADATA
//XDCADATA  DD DISP=SHR,DSN=FOOBAR.ADATALIB(&PGMNAME)
//XDCXCLUD  DD DISP=SHR,DSN=hlq.XDCV31.XDCSAMP(ZEXCLUDE)
//          DD DISP=SHR,DSN=your.exclusions
```

Note, **&PGMNAME** is an example PROC variable that conveys the name of the program to be assembled. YMMV.

For complete details, please see **HELP MAPS ADATA EXCESSADATA XDCADATA.**

Help UTilities XDCCall

The XDCCALL utilities are programs that are optionally used to create an environment for debugging your program.

There are four nearly identical utilities:

- **XDCCALL** is used to debug batch type programs that run **non-authorized.** Your program is entered with R1 pointing to a standard **PARM** field (if any). It can be used either in batch or in TSO.

- **XDCCALLA** is used to debug batch type and **System task** programs that run **authorized.** The program is entered with R1 pointing to a standard **PARM** field (if any). XDCCALLA can be used either in batch or in TSO.

  XDCCALLA is quite flexible in accommodating the needs of the program to be debugged. It supports unexpectedly sophisticated requirements, including these:

- If the program needs to run as a jobstep task, XDCCALLA can accommodate that.

- If the program is listed in your system's Program Properties Table (PPT) as requiring a special execution key, XDCCALLA can accommodate that.

For more information, see **HELP XDCCALL.**

- **XDCCMD** is used to debug formal TSO commands that run **non-authorized.** Your command is entered with R1 pointing to a TSO style **CPPL.**

- **XDCCMDA** is used to debug TSO commands that run **authorized.** Your command is entered with R1 pointing to a TSO style **CPPL.**

The four programs are nearly identical. In fact, they are all just individual entry points into the same program. There are only (and exactly) two differences amongst them.

1 - Their debugging session setups differ according to whether they are invoked authorized or non-authorized.

2 - The CMD[A] entries build a CPPL to pass to the target program, while the CALL[A] entries build a PARM field.

To use XDCCALL[A], you can use your program's normal startup proc, but you will have to make certain simple changes, and perhaps add certain //XDCnnnnn DD cards (see HELP DDNAMES) that XDCCALL[A] supports for various additional services.

For complete details, see HELP XDCCALL.

An alternative way to start a debugging session is to use either a dynamic hook or a static Hook. Then you can run your program without having to prepare it in any way. In particular, you won't even have to change its JCL. For details, see **HELP HOOKS.**

Help UTilities XDCMmef

The **XDCMMEF** utility is used to extract and compress mapping data for any or all load modules from one or more load libraries. It writes the data out to a portable export file called a **Module Mapping Extraction File** (or **MMEF** file for short).

In one execution, XDCMMEF can extract mapping data for load modules (including program objects) in any or all of the following:
- The Pageable Link Pack Area (PLPA)
- The current linklist
- Up to 50 libraries and library concatenations

The idea is that the extracted mapping data (ESD data mainly) can be used to assist dump/XDC in the analysis of an IPCS dump.

- When a software vendor receives a dump from one of its customers, the customer can also send along extracted mapping data for the load modules contained within that dump **without** having to send entire load modules! For a more detailed discussion, see **HELP DUMPS MAPPINGDATA BINDERMAPS.**

- When XDCMMEF extracts mapping data from a load module, it compresses and writes that data to an exportable sequential dataset that can be sent, along with the dump, to the dump examiner tasked with resolving the failure. The dump examiner can then use dump/XDC's **DEFINE MMEFDSN** command to make the customer's load module mapping data available for use during dump analysis.

- Subsequently, Z/XDC's **MAP** command will use the originating system's extracted mapping data (instead of searching the host system's load modules) for resolving mapping requests.

Running XDCMMEF

Normally, the **XDCMMEF** utility is executed in batch. The following is an example of the JCL you might use:

```
//<jobname> JOB <parameters>
//EXTRACT EXEC PGM=XDCMMEF,PARM='options'
//SYSPRINT  DD SYSOUT=*
//OUTFILE   DD DSN=<output.file.name>,
//             DISP=(,CATLG,DELETE),
//             UNIT=<unitname>,     [note: must be DASD]
//             VOL=SER=<volumename>,
//             SPACE=(CYL,(20,20),RLSE),
//             DCB=(RECFM=FB,LRECL=<whatever>,BLKSIZE=0)
```

The output dataset (//OUTFILE) will have RECFM=FB. The default LRECL will be 80, but it can be set to any reasonable value. A suitable BLKSIZE will be determined by the System.

Note, for various reasons the output file must be created on DASD, but when/if copied, it may be copied to any device type.

- The output file is a sequential dataset, and can safely be renamed. It can be shipped (for example, with a dump). If sending via FTP, then be sure to send it as a **binary** file.

Obtaining XDCMMEF

- It is expected that the XDCMMEF utility will be run by third party customers who are not, themselves, ColeSoft's customers. Accordingly, both XDCMMEF and its usage doc are publicly available and can be freely downloaded (with no obligation) from our website at **colesoft.com.** For more information, see **HELP SUPPORT ONLINE WEBSITE.**

More Information

Detailed XDCMMEF usage information is provided in a **.PDF** that also can be downloaded from **colesoft.com.**

Help UTilities XDCSymed

The **XDCSYMED** utility can be used to remove from an object file most redundant **SYM** data records. When properly used, significant file size reductions can be achieved!

SYM data is created by the high Level Assembler and is written (via //SYSLIN) to either a GOFF or classic format object deck.

When building a product from multiple assemblies, there typically will be a large amount of SYM data output that is duplicated in each assembly. The XDCSYMED utility, working in conjunction with certain otherwise innocuous source code changes, can detect this redundancy and eliminate it. For more information, see **HELP MAPS SYM EXCESSSYMDATA.**

To use XDCSYMED, you would add it to your assembly process in a step following the assembly step. You would cause the Assembler to write SYM data into an object (or GOFF) file that is allocated to a temporary dataset. Then an XDCSYMED step would read that temporary, remove redundant records and write the much diminished object (or GOFF) file out to a permanent member of an object library.

For more information, please see **HELP MAPS SYM EXCESSSYMDATA XDCSYMED.**

Help Virtmem

Z/XDC itself mainly runs in Primary Address Space Control Mode ("ASC-MODE", for short) and with both the Primary and Secondary Address Spaces set equal to the Home Address Space. When it needs to access other address spaces or data spaces, it briefly switches to access register mode (AR ASC-MODE), or it uses other mechanisms for the duration of those accesses.

Z/XDC can be used to debug programs running in any Address Space Control Mode and in any cross memory environment. Z/XDC, being an ABEND error recovery routine, can intercept any ABEND occurring in a user program running in those environments. When the user issues the TRACE or GO command to resume execution of his program, Z/XDC executes routines that restore the program’s ASC-MODE and cross memory environment prior to returning control to the user program.

Through Access Register ASC-MODE, a user program can directly access essentially any number of address spaces and data spaces in addition to its Home Space. Accordingly, a numeric virtual address no longer is sufficient to uniquely identify a specific location in storage. Now, further qualification is needed to indicate to which space a given numeric address belongs.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

CONCEPTS - Understanding address spaces, data spaces, and their accessibility to programs.

ACCESSLISTS - The role played by access lists in making address/data spaces available to programs.

ALET - Resolving Access List Entry Tokens against access lists. Also, special ALET values: 0, 1, and 2.

XDCACCESS - Addressing a program's spaces with Z/XDC.

MVSDOC - Where to find more information about z/OS supported address/data spaces and their access mechanisms.

Help Virtmem Concepts

Although programmers usually visualize an address space as being two dimensional, it really is only a one dimensional string of numbered locations starting at zero and ascending in a positive direction to some upper limit. However, now that programs can access multiple memories, a second actual dimension does come into play.

The spaces accessible to a user program can be thought of as a cluster of strings. The "major" string, if you will, is the program's Home Address Space. Therefore, a program's Home Address Space can be viewed as distinguishing a program's cluster of accessible spaces from other clusters of spaces accessible to other jobs in the system.

So, within a single job it is useful to imagine the accessible spaces as a two dimensional construct (a cluster of strings). However, system wide it is more useful to add a third dimension to the image. The accessible spaces of each separate job can be viewed as separate clusters of strings. Each cluster is identified or "named" by the owning job's Home Address Space.

The fact that some address/data spaces can be shared across multiple jobs does not particularly matter here. Such spaces can simply be viewed as being identical instances of strings occurring in multiple clusters. Further, if a cluster's Home Address Space itself is shared to another cluster, then it should be viewed as just another string in that other cluster, nothing special.

Help Virtmem ACcesslists

The cluster of address spaces and data spaces that a program can access through AR ASC-MODE ("Access Register Address Space Control Mode") is defined by various instances of two kinds of access lists that z/OS maintains:

- One kind of access list is named the "Primary Space Access List" ("PS-AL"). There is one PS-AL per address space. Therefore, a PS-AL provides a set of spaces that

is accessible to all tasks within an address space.

- The other kind of access list is named the "Dispatchable Unit Access List" ("DU-AL"). There is one DU-AL associated with every dispatchable unit of work (task mode **and** SRB mode) in every address space in the System. Therefore, a DU-AL can provide a unique set of spaces that are accessible only to individual tasks (and SRB routines) within an address space. In other words, two programs running under separate tasks but in the same address space can own data spaces that are private from each other.

Help Virtmem ALets

Spaces designated by entries in the PS-AL and DU-AL access lists become accessible to a program when ALETs ("Access List Entry Tokens") for these list entries are loaded into access registers. An ALET designates both which access list (PS-AL or DU-AL) to use and which entry in that list to use. When:
- A valid ALET is loaded into an access register (AR5 for example), and
- The current PSW's ASC-MODE flags are set to AR ASC-MODE, then

storage references made via the corresponding general register (R5 for example) are directed towards the address space or data space indicated by the access register.

By both hardware and operating system conventions, some ALETs have special meanings:
- An ALET value of 0 (X'00000000') always refers to the Primary Address Space. This is an automatic convention of the hardware. Neither the PS-AL nor the DU-AL is checked.
- An ALET value of 1 (X'00000001') always refers to the Secondary Address Space. This also is an automatic convention of the hardware.
- An ALET value of 2 (X'00000002') always refers to the Home Address Space. This is not a hardware convention. An ALET value of 2 causes the hardware to use the 2nd entry in the current DU-AL. It is a z/OS convention that this 2nd entry will always designate the Home Address Space.

Collectively, these special ALET values (0 1 and 2) are called either "generic ALETs" or "keyword ALETs".

When a program ABENDs and passes control to Z/XDC, Z/XDC maintains information about two distinct execution environments associated with the program: the error level environment and the retry level environment. This information includes a complete set of such execution time characteristics as register values, PSW contents, Primary and Secondary Address Space designations, etc. The execution characteristics of the retry level environment may be either the same as or quite different from the error level environment. These environments are discussed in depth in HELP EXECUTIONLEVELS.

In particular, it is quite possible for the Primary and Secondary Address Spaces to be different between the retry level environment and the error level environment. Z/XDC takes this fact into account when resolving the generic ALET values of 0 (Primary) and 1 (Secondary):
- When an ALET is taken from an error level access register, ALET values of 0 and 1 are resolved to the Primary and Secondary Address Spaces relative to the error level environment.
- When an ALET value is obtained from any other source, Z/XDC resolves the values of 0 and 1 to the Primary and Secondary Address Spaces relative to the retry level environment.
- The Home Address Space is the same for both the error level and retry level

environments, so an ALET value of 2 is always unaffected by the retry vs. error level environment. It always resolves to the Home Space.
- All other ALET values are always resolved via the PS-AL and the DU-AL available to Z/XDC running under the current task in the Home Address Space.

Help Virtmem Xdcaccess

Z/XDC permits users to display and alter storage within any address/data space that the user's program has access to. Within an address expression, a particular space within a program's cluster of address and data spaces can be designated via a built-in function named **ALET(...).** This function's operand designates an **ALET** that is used to resolve subsequent storage references within the address expression.

The **ALET(...)** function can be used any number of times in an address expression, thus it is possible for a single address expression to follow control block chains that wander through multiple spaces.

While the ALET(...) function allows Z/XDC to select a particular address/data space within a program's cluster of spaces, the cluster itself is selected either by the **SET ASID** command or by another built-in function named **ASID(...).** This command and this function can be used by authorized users to cause Z/XDC to target another job's cluster of spaces instead of the current job's cluster. This is referred to in Z/XDC documentation as **Foreign Address Space Mode** or **FASM,** for short.

- The **SET ASID** command can be used to set (or change or terminate) Foreign Address Space Mode for all subsequent commands.

- The **ASID(...)** built-in function can be used to set (or change or clear) Foreign Address Space Mode for only the remainder of the current address expression.

When Z/XDC is not running in Foreign Address Space Mode, then it is running in **Local** Address Space Mode **(LASM).** This means that Z/XDC can display, zap, and otherwise manipulate the cluster of address/data spaces that are accessible to the current job, TSO session, or system task within which Z/XDC is running.

Note that the contents of another address space might be accessible to Z/XDC **both** via Foreign Address Space Mode and **via an ALET** while in Local Address Space Mode. Access to another address space via ALETs requires cooperation from the user program being debugged because it is up to the user program to make the other address space's ALET available for use; Z/XDC cannot do that by itself. FASM mode, on the other hand, does not require any cooperation from or even awareness by the user program; Z/XDC itself performs all the work necessary to make the access.

Accordingly, in order to use FASM mode, Z/XDC must be running authorized, and the user's access to each specific address space must be permitted by system security. On the other hand, these checks are not necessary for such accesses made via the ALET(...) function while in LASM mode.

LIMITATION: Currently, while in Foreign Address Space Mode, only the Home Address Space of the targeted cluster is accessible to Z/XDC; other spaces in the cluster are not. In other words, use of the ALET(...) function is not supported while in FASM mode.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

FASM - More information about Foreign Address Space Mode.
LASM - More information about Local Address Space Mode.

Help Virtmem Xdcaccess Fasm

For a comparison of FASM mode with cdf/XDC, see HELP XDCSRVER CDF FASM.

When Z/XDC is in Foreign Address Space Mode ("FASM mode", yeah I know it's redundant), Z/XDC's various FORMAT, DISPLAY, LIST, VERIFY, ZAP, HOOK and POST commands are targeted towards the address/data space cluster owned by some other job, TSO session, or system task ("job" for short) currently running elsewhere.

FASM mode is a very powerful method for exploring, investigating, and debugging other jobs. Z/XDC runs within your own TSO session, not within the targeted job, so Z/XDC requires no cooperation from that job. In fact, the targeted job can be running normally, or it can even be "dead": locked in a wait state or in a closed loop! All that is required is that the targeted job's Home Address Space exist in the same z/OS system within which your Z/XDC debugging session is running.

FASM mode is established, changed, or terminated by the SET ASID command. Once established by this command, it generally remains in effect until changed or terminated by another SET ASID command.

FASM mode can also be overridden for the duration of an address expression by the **ASID(...)** built-in function. This function can be used to establish, change, or terminate FASM mode:
- For the duration of only a single Z/XDC command,
- Or even more briefly, for the duration of only part of an address expression.

The ASID(...) function can be used one or more times in any address expression.

FASM mode can be used only when Z/XDC is running authorized. In addition, a system security call is made to check whether or not the specific user attempting to use FASM mode has permission to use FASM mode against the specific job, TSO session, or system task that he is targeting. See HELP SECURITY FASM for more information.

Except for use of the various storage altering commands (ZAP, HOOK, POST, TRAP, etc.), FASM mode is a completely noninvasive and nondisruptive mechanism. Nearly all of Z/XDC's information processing and display commands can be used. The following is a partial list of commands that are available while in FASM mode:

FORMAT - Formats and displays storage.
DISPLAY - Displays storage in a raw hex-text (dump like) format.
LIST TASKS - Displays the subtasking tree for an address space.
LIST RBS tcbaddress - Displays the Request Blocks currently queued to a given TCB.
LIST PSW rbaddress - Displays the resume PSW associated with a given Request Block.

LIST REGS rbaddress - Displays the general registers **associated** with a given Request Block.

LIST AREGS rbaddress - Displays the access registers **associated** with a given Request Block.

LIST LSTACK --- - Displays the linkage stack entries for a specified task.

LIST ESTAES --- - Displays a given task's SCB queue to show information about the task's ABEND error recovery environment.

LIST PGMS - Displays in various ways the load module queues (job pack queues, load lists, etc.) within an address space.

LIST RSA - Displays register save area chains.

LIST SUBPOOLS - Displays subpools and other storage allocation information for any aspace.

LIST TIOT --- - Displays the ddnames allocated for any task.

EQUATE - Defines individual equate symbols for labeling storage in an address space.

MAP - Loads module and csect maps for load modules contained in an address space.

DMAP - Loads dsect maps into storage.

USING - Assigns dsect maps to locations in an address space.

VERIFY - Verifies that a given storage location contains expected data.

SET PSW AMODE - This changes the addressing mode of the retry level PSW in the **Home** Address Space, not the foreign address space. This command is supported in FASM mode only because the local retry level PSW's AMODE can affect certain types of ZAP commands.

In addition, all commands that manipulate Z/XDC objects (e.g. LIST/DELETE EQUATES) and that deal with debugging session characteristics (PROFILE, SET COLORS, ISPF, etc.) or processing (e.g. READ) also can be used, regardless of whether or not Z/XDC is in FASM mode.

The following potentially "disruptive" commands also can be used while in FASM mode:

ZAP - Changes data in storage.

HOOK - Dynamically creates a Z/XDC debugging session (reachable via cdf/XDC) in the targeted address space.

POST - Posts an ECB.

Also, the various breakpointing commands (AT, ATX, TRACE, and TRAP) can be used, but their use is beneficial only when the target address is within a PC routine that (a) will be called from the current address space and (b) will establish local Z/XDC protection prior to execution reaching the breakpoint. See HELP DEBUGGING PCROUTINES for more information.

Use of these storage altering commands is controlled by a security call separate from that used for the information display commands. See HELP SECURITY FASM for more information.

The following is a partial list of the commands that **cannot** be used in FASM mode:

- THE GETMAIN/FREEMAIN commands
- THE LOAD/DELETE MODULE commands
- ZAP PSW ...
- ZAP Rn ...

- Versions of SET PSW other than the SET PSW AMODE command
- The deferred breakpointing commands: ADEFERRED and TDEFERRED

- Any reference to a **default** PSW or register: In Z/XDC the default PSW and registers are those owned by the "current task". However, from the point of view of Z/XDC running in your TSO session, there is no such thing as a "current task" in another address space. The only task that's "current" is Z/XDC itself running in the Home Address Space. Accordingly, the following commands cannot be supported:

  - LIST PSW without giving an RB address.
    Use the "LIST PSW rbaddress" form of this command.

  - LIST REGS without giving an RB address.
    Use the "LIST REGS rbaddress" form of this command.

  - LIST AREGS without giving an RB address.
    Use the "LIST AREGS rbaddress" form of this command.

  - LIST RBS without giving a TCB address.
    Use the "LIST RBS tcbaddress" form of this command.

  - LIST LSTACK without giving a TCB address.
    Use the "LIST LSTACK tcbaddress" form of this command.

  - LIST ESTAES without giving a TCB address.
    Use the "LIST ESTAES tcbaddress" form of this command.

Regarding debugging background jobs: When **cdf/XDC** is being used to allow a TSO user to debug a batch job or other address space, the Z/XDC running in the user's TSO address space is communicating with another Z/XDC running in the targeted batch job. The TSO copy of Z/XDC is only performing terminal display tasks; it is the batch job's copy of Z/XDC that is performing the debugging work being requested by the user. Accordingly, a debugging session against a batch job using cdf/XDC is in Local Address Space Mode, **not** in FASM mode. Accordingly, cdf/XDC debugging sessions can use the full range of Z/XDC's commands. They are not subject to the restrictions that FASM mode is.

Subtopics Index

FASM mode also can be used to display and alter **real** storage. For more information, type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

REAL - Using Foreign Address Space Mode to access real storage.

Help Virtmem Xdcaccess Fasm Real

Foreign Address Space Mode can be used, not only to access other address spaces, but also to access **real** storage as well. To do so, either:

- Use the REAL(...) built-in function in your address expressions.
- Refer to real storage by the name "REAL" in the ASID(...) built-in function.
- Refer to real storage by the name "REAL" in the SET ASID command.

Examples:

FORMAT REAL(IEFBR14)
The virtual storage location of IEFBR14 is determined. Then the real storage location at which that virtual location currently resides is determined and displayed. Warning: the real address of virtual pages cannot be relied upon unless the virtual page is page fixed **and** the owning address space is nonswappable! In all other cases, the correspondence between a page's virtual and real addresses can change rapidly, randomly, and unpredictably. In fact, it can change even **during** the time that it takes Z/XDC to generate its display!

FORMAT IEFBR14~ASID(REAL)
This determines the virtual address of IEFBR14 and then displays the real storage location having the same numeric address. This will likely **not** be the real storage location of IEFBR14.

DMAP XDCMAPS.ASCB
DMAP .ASTE
USING ASCB @ASCB F
USING ASTE .ASCBASTE? F
DISPLAY .ASTEAETO?~ASID(REAL)
The DMAP and USING commands load dsect maps and assign then to represent the Target Address Space's Address Space Control Block and Address Space Second Table Entry, respectively. The DISPLAY command then displays that address space's Authorization Table, located in real storage.

SET ASID(REAL)
FORMAT 0
The SET ASID command sets the default Target Address Space to real storage. The "FORMAT 0" command displays the contents of real location zero.

Help Virtmem Xdcaccess Lasm

When Z/XDC is not in Foreign Address Space Mode, it is in Local Address Space Mode, or "LASM mode" for short. When in LASM mode, Z/XDC's **full** capabilities are available for debugging the user program in whose Home Address Space Z/XDC is running.

When Z/XDC is in LASM mode, it is able to map, format, display, and alter both the user program's Home Address Space and any other address space or data space to which the user program has access via ALETs. If the user program does not have access to another space via ALET, then Z/XDC also does not have access to that space while in LASM mode.

When cdf/XDC is being used to allow a TSO user to debug a batch job or other address space, the Z/XDC running in the user's TSO address space is communicating with another Z/XDC running in the targeted batch job. The TSO copy of Z/XDC is only performing terminal display tasks; it is the batch job's copy of Z/XDC that is performing the debugging work being requested by the user. Accordingly, a debugging session against a batch job using cdf/XDC is in LASM mode, **not** in Foreign Address Space Mode. Accordingly, cdf/XDC debugging sessions are not subject to the restrictions that FASM mode is.

Help Virtmem Mvsdoc

Programs running in z/OS systems have access to numerous address spaces and data spaces ("spaces", for short) via many different mechanisms. z/OS permits programs to access a "Home" Address Space, a "Primary" Address Space, and a "Secondary" Address Space. These three address spaces are identified by page table, segment table or region table origin pointers located in various control registers. Depending upon the user's program, these three spaces may be all the same, all different, or nearly any combination in between.

In addition, programs can access arbitrary other address spaces as well as a new kind of storage called a "data space" via Access List Entry Tokens ("ALET"s) located in access registers.

These various spaces may or may not be accessible to the user's program depending upon the Address Space Control Mode ("ASC-MODE": Primary, Secondary, Access register or Home) that is set within its PSW and depending upon whether or not a particular entry represented by an ALET currently is or is not present in an Access List (PS-AL or DU-AL).

It is well beyond the scope of this Built-in Help to explain how all this works. You should refer to the following IBM manuals for that information.

- Principles of Operation, SA22-7201.
- Extended Addressability Guide, GC28-1769.
- Assembler Services Guide, GC28-1762.
- Assembler Services Reference, GC28-1910. (macros)
- Authorized Assembler Services Guide, GC28-1763.
- Authorized Assembler Services Reference, GC28-1764 through GC28-1767. (macros)

Help XDCCAll

XDCCALL is a TSO command or batch program that creates an environment in which your program can be debugged using Z/XDC. XDCCALL can be invoked in several ways:

- In **ISPF,** Z/XDC's **Startup Panel** can be used to start a debugging session in an environment created by XDCCALL. See HELP DEBUGGING ISPF PANEL for more information.

- In **TSO,** XDCCALL can be started up as a TSO command with an operand providing the name of the program to be debugged. Example:

XDCCALL programname parmfielddata

- In the background **batch,** XDCCALL can be started via some pretty simple JCL:
  //STEP EXEC PGM=XDCCALL,PARM='programname parmfielddata'

- In the batch, **PARMDD=ddname** also is supported. By this means, PARM field data up to 32,742 bytes long can be passed to the program to be debugged. Note, similar to the PARM= case, the data in the PARM data dataset must start with the name of the program to be debugged.

Regardless of the method used, the **programname** (along with its trailing blanks) is stripped out before the target program sees the **parmfielddata.**

How it Works

In brief, XDCCALL attaches your program as a subtask with Z/XDC established as that subtask's **ESTAI** routine. This means that Z/XDC will be given control for any ABEND (or breakpoint) that occurs either within your program or within any subtasks that it might create.

There is a catch: If your program does anything to create its own ABEND recovery environment, then something has to be done to reestablish Z/XDC as a **newer** recovery routine than yours so that Z/XDC will see ABENDs and breakpoints before your code does. Fortunately, the **HOOK** command provides a very easy way to deal with this. See HELP COMMANDS HOOK for more information.

Z/XDC Clones and XDCCALL's Name

When Z/XDC's name has been changed to some name other than **XDC,** then XDCCALL's name also must be changed to match the new **clone** name. So for example, if Z/XDC's name has been changed from XDC to **V31,** then XDCCALL's name must also be changed from XDCCALL to **V31CALL.**

As a reminder of this, XDCCALL from here on out will be referred to in this doc as **xxxCALL.** For more information, see HELP XDCCLONES.

xxxCALL and its Several Alternate Names

xxxCALL has several **alias** names (altername names) that affect certain basic aspects of the debugging environment that it creates. For example, if you wish to debug a normal program running non-authorized (problem key, problem state, AC=0, etc.), then use the name **xxxCALL.** On the other hand, if you want to debug an **authorized** program (AC=1, System Security permitting) then use the name **xxxCALLA.**

When Z/XDC is running under a clone name **(xxx)** other than **XDC,** then XDCCALL and its various aliases must be invoked via that same clone name: xxxCALL, xxxCALLA, xxxCMD, and xxxCMDA. For information about renaming Z/XDC, see HELP XDCCLONES.

Altogether, xxxCALL/xxxCMD has four different names:

xxxCALL - When this name is used, xxxCALL/xxxCMD considers your program to be

a **batch type** program (ie, **not** a TSO command). Accordingly, xxxCALL passes parameters, if any, to the user's program in the form of an OS-style **PARM=** field pointed to (indirectly) by R1. This is regardless of whether xxxCALL is invoked in the batch or in TSO.

xxxCMD - When this name is used, xxxCALL/xxxCMD considers the user's program to be a **TSO command.** Accordingly, xxxCMD passes parameters, if any, to the user's TSO command in the format of a command buffer pointed to by a **CPPL** (pointed to by R1).

xxxCALLA - Using this name causes xxxCALL/xxxCMD to behave the same way as when the xxxCALL name is used except that the user's program is executed **authorized (AC=1)** instead of non-authorized.

xxxCMDA - Similarly, using this name causes xxxCALL/xxxCMD to behave the same way as when the xxxCMD name is used except that the user's TSO command is executed **authorized (AC=1)** instead of non-authorized.

Except for the differences described above, the treatment of the user's program by xxxCALL/xxxCMD remains **identical in all respects** regardless of by which name xxxCALL/xxxCMD is invoked.

In particular, if a program **does not make use** of either a **PARM field** or a **CPPL,** then it is **irrelevant** whether it is invoked via xxxCALL or xxxCMD.

Where and How to Run xxxCALL/xxxCMD

xxxCALL/xxxCMD can be executed either in TSO or in a background batch job or as a system task:

- When invoked in TSO, xxxCALL/xxxCMD should be invoked as a normal TSO command. For example: **XDCCALL IEBGENER SDB=LARGE**

- When invoked in the batch, you must use JCL:
  - With xxxCALL/xxxCMD's name provided by the **EXEC PGM=** parameter,
  - With the name of the program to be debugged provided by the start of the **PARM='...'** parameter.
  - And the program's own PARM field data (if any) following in the PARM= field.

  Example: **//STEPNAME EXEC PGM=XDCCALL,PARM='IEBGENER SDB=LARGE'**

PARM fields or Command Buffers

xxxCALL/xxxCMD is kind of a middleman. It receives either a PARM field or a command buffer from its caller (usually the System), and it constructs a new PARM field or new command buffer to be passed on to the program or TSO command to be debugged. Here are the details:

- When you invoked xxxCALL (or xxxCMD) in ISPF or TSO, all command operands that you provide are passed to xxxCALL/xxxCMD via a TSO style **Command Buffer.**

- When you invoked xxxCALL (or xxxCMD) in the batch, You must provide its operands via a z/OS style **PARM field.**

- Regardless of whether xxxCALL/xxxCMD receives its operands via a PARM field or a command buffer, it will build either a new PARM field or new command buffer for passing appropriate operands onwards to the program or TSO command to be debugged.

- Whether xxxCALL/xxxCMD builds a PARM field or a command buffer depends solely upon whether xxxCALL/xxxCMD is running as xxxCALL (builds a PARM field) or as xxxCMD (builds a command buffer).

- Whether xxxCALL/xxxCMD is being invoked in TSO or in the batch is not relevant.

Mixed Case PARM Data or Command Operands

When xxxCALL/xxxCMD receives mixed case data via a PARM field or a command buffer, it preserves the casing when forwarding data on to the program to be debugged. The problem is... keeping the system from upcasing mixed case data prior to xxxCALL/xxxCMD receiving it. Here are the details:

- When xxxCALL/xxxCMD is invoked via JCL in the batch, there is no problem. z/OS preserves the casing of PARM field data, so xxxCALL/xxxCMD simply passes it onwards to the target program.

- Similarly, when xxxCALL/xxxCMD is invoked either from TSO's READY prompt or from ISPF's Command Shell (=6), again there is no problem. TSO preserves the command line casing and so does xxxCALL/xxxCMD. They simply pass the data onwards as-is to the target program.

- The problems come in when invoking xxxCALL/xxxCMD via Z/XDC's Startup Panel in ISPF. Under the covers, the Panel invokes a CLIST (XDCCLIST), and that CLIST eventually invokes xxxCALL/xxxCMD via ISPEXEC SELECT CMD(yadayada). Unfortunately, the SELECT Service unconditionally upcases the yadayada.

  But I've got some magic up my sleeve. The CLIST provides a way to save the original mixed case data where xxxCALL/xxxCMD can find it, thus enabling xxxCALL/xxxCMD to restore the mixed case PARM field data or command operands before forwarding it on to the program to be debugged.

  But upcasing is optional. The Z/XDC Startup Panel has an input field whereby the user can select whether or not casing is to be preserved.

Security Concerns

Programs that are to be debugged authorized (via xxxCALLA or xxxCMDA) need not themselves be APF authorized. Nor do they need to reside in authorized libraries. Nevertheless, they **will run authorized!** So the use of xxxCALLA/xxxCMDA does create serious security concerns. Note, however, that the availability of the xxxCALLA and xxxCMDA commands can be restricted both by system security routines and by decisions made at Z/XDC installation time. For more information, see the several security related topics starting at **HELP SECURITY.**

Proxy Tasks

When debugging SRB routines and other code protected by an FRR routine, Z/XDC itself needs to run as an FRR, and when it does, it needs **Proxy Tasks** into which it can schedule the bulk of its processing (referred elsewhere as Z/XDC's **Remote Phase).** This all is discussed in excruciating detail in HELP DEBUGGING FRR, so I'll not go further into it here.

But xxxCALLA does have a role to play regarding Proxy Tasks. As a side effect of creating the debugging environment, xxxCALLA (as well as xxxCALL) can also create **Formal Proxy Tasks** for eventual use during FRR-mode debugging.

By default, xxxCALLA will automatically create **two** Formal Proxy Tasks, but you can use a **//xxxFPTnn DD DUMMY** allocation to override that default and create anywhere from zero to ten Formal Proxy Tasks. See **HELP DDNAMES FPTNN** for more information.

(Note, once Z/XDC proper is up and running, Proxy Tasks can also be created by the **SET PROXYTASKS** command.)

(Further, when Z/XDC is running authorized, the SET PROXYTASKS command can be used to create Proxy Tasks in **other address spaces** where Z/XDC might **not yet** be running!)

The Z/XDC Startup Panel

In ISPF, when a debugging session is started by use of Z/XDC's Startup Panel, when either option 1 or 2 is selected, then xxxCALL (option 2) or xxxCMD (option 1) is used to start the debugging session. For more information, see HELP DEBUGGING ISPF PANEL.

xxxCALL (and **all** of its aliases) can be invoked either in TSO or in the batch. Regardless of its environment, xxxCALL will determine **by inspection** what its caller environment is and whether it is receiving from its caller an OS style PARM field or a TSO style CPPL.

In either case, xxxCALL will convert the information from its caller (batch or TSO) to a standard internal form and then reconvert it to the form expected by the batch program or TSO command to be debugged.

Note that if the xxxCMD or xxxCMDA name is used in the batch, then the CPPL that xxxCALL must generate for the user's program will be incomplete: It will have pointers to a UPT and a CBUF, but not to a PSCB or an ECT.

Examples Regarding xxxCALL Names and Your Program's PARM Data

XDCCMD LISTA STATUS
XDCCMD ISPF
XDCCMDA SEND 'PLEASE CANCEL MY JOB'
These examples all invoke various TSO commands within a Z/XDC debugging environment. The **SEND** command is invoked authorized, while the **LISTA** and **ISPF** commands are invoked non-authorized. In each case the operands of the XDCCMD(A) command **(LISTA STATUS,** for example) are placed into a TSO-type command buffer, and a CPPL is build that, among other things, points to the command buffer.

Before passing control to the target command, R1 is made to point to the CPPL.

V31CALL IEHLIST LINECNT=55
V31CALL LKED RENT,REUS,LET,NCAL
V31CALLA IEBCOPY SIZE=1024K

These examples all invoke various background type programs within a Z/XDC debugging environment. In this case, a clone of Z/XDC is used that has been renamed from XDC to **V31.**

The **IEBCOPY** program is invoked authorized, while the **IEHLIST** and **LKED** programs are invoked non-authorized. In each case the parameters for the target program **(RENT,REUS,LET,NCAL,** for example) are placed into an OS-style **PARM=** field, and R1 is made to point to a pointer to the PARM= field.

//LOGON EXEC PGM=XDCCALLA,PARM='IKJEFT01 %LGONCLS'
In this example, XDCCALLA is invoked via JCL. The program to be debugged is **IKJEFT01.** By the time that program receives control the string "IKJEFT01" will have been deleted from the PARM field, so the program will just see the equivalent of **PARM='%LGONCLS'.** Note, IKJEFT01 will receive control as an **APF authorized (AC=1)** program.

//MYPGM EXEC PGM=XDCCALL,PARM='sendmsg It's Closing Time'
In this example, the user written program named **SENDMSG** is started up within a debugging session, and its PARM field data is **" It's Closing Time".** Notice that:
- The casing of the PARM data was preserved.
- The program name ("sendmsg") along with **only one** of its trailing blanks was stripped out from the PARM field data.
- The name of the program ("sendmsg") was upcased before XDCCALL searched System Libraries to find it.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

ENVIRONMENT - The debugging environment
INTERFERINGESTAES - Keeping Z/XDC first in line to see ABENDs.
EXITS - Special exit routines
TASKLIB - Task library support
JOBSTEPTASKS - Setting up your target program to run as a jobstep task
DDNAMES - Special ddnames

Help XDCCAll ENvironment

Preliminary Notes

The term **xxxCALL** refers to the XDCCALL program. The **xxx** denotes the fact that when Z/XDC is renamed to some three character name (referred to herein as **xxx)** other than **XDC,** then XDCCALL must also be similarly renamed.

The xxxCALL program has three alias names: xxxCALLA, xxxCMD and xxxCMDA. The name **xxxCALL** will be used herein to collectively refer to all four names.

For more information, see HELP XDCCALL.

Passing Control from xxxCALL to Z/XDC and Then to Your Program

Once xxxCALL (or xxxCMD, etc.) has set up the debugging environment, it passes control to Z/XDC just before allowing the target program to start (unless a **//XDCQUICK** allocation is present, in which case the initial stop at Z/XDC is bypassed, and your program begins execution immediately).

If Z/XDC does receive initial control, then you are given the chance to issue Z/XDC commands before your program actually begins execution. Generally, this is a good time to load maps and set breakpoints and do whatever else you want before letting your program start execution.

If, however, you have provided an //xxxCMDS file (see HELP XDCCALL DDNAMES for more information), then the commands contained within that file are executed first. Those commands may or may not change the environment described below.

The Entry Environment Created for Your Program by xxxCALL

In any case, in the absence of an //xxxCMDS file, the initial environment presented both to you and to your program is as follows:

- The target program is running as a **subtask** of xxxCALL (or xxxCMD, etc.). This will be seen if you issue the **LIST TASKS** and **LIST RBS** commands.
- If xxxCALL (or xxxCMD, etc.) was requested to provide a **task library,** then that library will now be used for any LOAD, LINK, ATTACH, and XCTL macros that your program might issue. (See HELP XDCCALL TASKLIB for more information.)
- The current instruction, as shown by the **WHERE** command, is your program's first instruction.
- Your program's addressing mode (AMODE) is set to 24, 31 or 64 according to your program's AMODE attribute as set by the Binder.
- xxxCALL sets up the general registers with the following initial values:
  - **R15:** For AMODE(24) and AMODE(31) programs, R15 points to your program's entry address. For AMODE(64) programs, R15 contains the value **X'FFFFF00x'** (where **x** can be 0, 2 or 4. For more information, refer to IBM's **z/OS MVS Programming: Assembler Services Guide** [SA22-7605], and then

search for the string **FFFFF002).**

- **R14:** This register points to a return address that your program should use when it completes. Doing so returns execution to a breakpoint (a DEAD trap, actually) that causes control to pass back to Z/XDC one last time before the debugging session ends. Notes:
  - This return address will always be located in 24-bit storage.
  - If your program does not return via R14 but instead terminates via an SVC 3, then this last breakpoint is bypassed.

- **R13:** This register points to a system provided save area buffer pointed to by the current TCB's **TCBFSA** field. This buffer has the following characteristics:
  - It is 144 bytes long.
  - So it is long enough to contain either a standard, 72-byte register save area, or a 144-byte, F4SA (format-4) register save area.
  - Initially, the 144-byte save area buffer has been entirely zeroed by the system.

- **R0:** The contents that xxxCALL received when it first received control from the System is passed through unchanged to your program.

- **R1:** This register points either to a TSO-style **CPPL** (Command Processor Parameter List) or to an OS-style **PARM** field, depending upon whether the debugging session was started by XDCCMD(A) or XDCCALL(A). A **FORMAT R1! DATA** command will show this. Notes:
  - The PARM field (or command buffer) may contain mixed case data.

  - When running in the batch, both XDCCALL and XDCCALLA support long PARM fields up to 32,751 bytes long (including the target program name).

    But by the time the target program sees the PARM field, the target program name has been stripped out, so the longest PARM field that can reach the target program is only 32,742 bytes.

    It is possible that XDCCALL[A] may truncate the given PARM field data, but when it does, it issues (to SYSLOG) a warning message (DBC598W) about this.

  - When invoked within TSO, neither XDCCALL nor XDCCALLA support any equivalent of PARMDD=ddname.

- **R2-R12:** **!!!WARNING!!!** These registers are **NOT** guaranteed to be zeroed! They may contain random information.

XDCCALL and the Program Properties Table (PPT)

If your program has an entry in the PPT that assigns to your program an execution key other than 0 or 8, then XDCCCALL cannot be used to debug your program when running within TSO.

The problems that occur in TSO do not, however, occur in the batch, so XDCCALL can be used just fine with PPT-listed programs in the batch.

For more detailed information, see HELP MESSAGES DBC529.

Help XDCCAll Interferingestaes

Preliminary Notes

The term **xxxCALL** refers to the XDCCALL program. The **xxx** denotes the fact that when Z/XDC is renamed to some three character name (referred to herein as **xxx)** other than **XDC,** then XDCCALL must also be similarly renamed.

The xxxCALL program has three alias names: xxxCALLA, xxxCMD and xxxCMDA. The name **xxxCALL** will be used herein to collectively refer to all four names.

For more information, see HELP XDCCALL.

Coping with Interfering Recovery Environments

xxxCALL is a very convenient device for setting up a debugging session in an easy way. However, there is a catch: If your program does anything to create its own ABEND recovery environment (ie:
- If it issues any ESTAE, ESTAEX or SETFRR macro,
- Or if it issues an ATTACH macro with an ESTAI= operand,
- Or if it issues an IEAMSCHD macro with an FRRADDR= operand,
- Or if it sets up an ARR for a PC routine),

Then if an ABEND or breakpoint occurs in code that is protected by your recovery routines, then your recovery routine will receive control **ahead of** Z/XDC, and that usually results in unexpected and totally confusing **havoc!**

There is an easy solution for this catch. If you wish to debug code for which you have created your own recovery environment, then you can, on the fly, set a **hook** in the code at the point at which you wish to start debugging. A hook is like a breakpoint in that it intercepts program execution. But in addition, it creates or recreates a new Z/XDC environment so that breakpoints and ABENDs will be seen by Z/XDC **before** being seen by your recovery routine!

Setting a hook is trivially easy! Just use the **HOOK** command to create a hook in the code of your choice.

Hooks can be inserted into nearly any kind of code running pretty much anywhere. This includes:
- Problem mode code
- Authorized code
- Taskmode code
- SRB routines
- ESTAE protected, ESTAI-protected, ARR-protected, etc-protected code
- FRR protected code
- PC routines
- Locked code
- Disabled code
- PRIMARY and AR ASC-MODE code
- SECONDARY and HOME ASC-MODE code (There are special considerations in this case - refer to the HOOK command).
- Code running in Foreign Address Spaces

- Whatever!

For more information, see HELP HOOKS.

Help XDCCAll EXits

Preliminary Notes

The term **xxxCALL** refers to the XDCCALL program. The **xxx** denotes the fact that when Z/XDC is renamed to some three character name (referred to herein as **xxx)** other than **XDC,** then XDCCALL must also be similarly renamed.

The xxxCALL program has three alias names: xxxCALLA, xxxCMD and xxxCMDA. The name **xxxCALL** will be used herein to collectively refer to all four names.

For more information, see **HELP XDCCALL.**

User Writable Exits Supported by xxxCALL

Z/XDC supports a number of user writable exit routines. (See **HELP EXITS** for more information.) Normally, the user must make Z/XDC aware of the presence of these exits by placing pointers to them into a control block that is passed to Z/XDC via the **PARAM=** operand of the ESTAE macro used to establish Z/XDC processing. xxxCALL (et al.) however, provides another mechanism for making exit routines available to Z/XDC.

During its setup processing, xxxCALL, among other things, looks for two special load modules, named xxxUCI and xxxXITS. If either or both of these modules are found, then they will be loaded into storage and pointers to them will be made available to Z/XDC.

xxxUCI, if present, must contain a locally written **User Communications Interface** to be used by Z/XDC in place of its normal TPUT/TGET (or WTO/WTOR) routines for communicating with the user's terminal. For more information, see **HELP USERINTERFACES UCI.** Also, a sample program that makes use of the User Communications Interface is named **SRCONLHG** and can be found in a library whose factory default name is hlq.XDCV31.XDCASM. (You will have to ask your Systems Programmer for the library's actual name at your data center.)

xxxXITS, if present, must contain one or more of the following "general" exit routines:

- User Commands Exit
- Initialization Exit
- Resumption Exit
- Termination Exit
- User SVC Exit

See **HELP EXITS** for more information. Also, a sample program that makes use of the User Commands Exit can be found in two members of the hlq.XDCV31.XDCASM library. Their names are **SRCXITSR** and **SRCXUCMD.**

xxxCALL (and friends) will search for these modules:

- In **//xxxLIB** (if provided)
- In a **task library** (if available)
- And the locations found in the System's normal load module **search order:**
    - //ISPLLIB
    - //STEPLIB
    - //JOBLIB
    - PLPA
    - Linklist libraries

For more information, see **HELP XDCCALL TASKLIB.**

Help XDCCAll Tasklib

Preliminary notes

The XDCCALL program has three alias names: XDCCALLA, XDCCMD and XDCCMDA. The name **XDCCALL[A]** will be used herein to collectively refer to all four names.

For more information, see **HELP XDCCALL.**

User Program Task Library Setup

Normally, when looking for the program or command to be debugged, XDCCALL[A] searches according to IBM's normal search order which is as follows:

- The Home Address Space's **Job Pack Area** (for reentrant or reusable modules already loaded into the private area).
- **Task Libraries** (if any) established by any **ancestor task** above XDCCALL[A]'s task.
- The current job or session's **STEPLIB** or **JOBLIB** library.
- The System's current **Link Pack Queue** (for common storage modules located in the DLPA, the MLPA or the FLPA).
- The System's **Pageable Link Pack Area** (PLPA).
- The System's **linklist** libraries (starting with SYS1.LINKLIB).

[Note, you can use the **LIST SEARCHORDER** command to display the execution environment's current load module Search Order.]

If, however, the program that you wish to debug is not located in any of these places, then you may add a **task library** to the **front** of the search order.

A task library is like a STEPLIB, but its scope is limited to a particular task and all its descendent tasks.

When a **//TASKLIB DD** is present, the **XDCCALL[A]/CMD[A]** utilities will set that ddname as the task library for the program to be debugged. For detailed information about task libraries in general, please see **HELP XDCCALL TASKLIB TASKLIBRARIES.**

For specific information about **XDCCALL[A]'s** support of task libraries, just read on.

How to set Up a Task Library for Your Debugging Session

To use XDCCALL[A]'s support for task libraries, proceed as follows:

- Prior to issuing the XDCCALL[A] command (or the XDCCMD[A] command, etc.), allocate the library(ies) containing the desired program(s) to ddname **TASKLIB.** (Concatenations are permitted). XDCCALL[A]/CMD[A] will then establish the TASKLIB allocation as the task library for the subtask in which XDCCALL[A] runs the program to be debugged.

- Alternatively, allocate the desired library(ies) to **any ddname** of your choice. Then issue the XDCCALL[A] (et al.) command with the chosen ddname prefixed to the target program name and delimited with a colon (:). Example:

  TSO ALLOC F(SPCLTLIB) DA(LOADLIB) REU SHR
  XDCCMD SPCLTLIB:LISTLOG YESTERDAY

  - In this example:
    - **LISTLOG** is a locally written TSO command residing in a library named **LOADLIB** under my own TSO userid.
    - The **ALLOC** command allocates that library to ddname **SPCLTLIB.**
    - The **XDCCMD** command then invokes the **LISTLOG** program out of that library.
    - Further, the library is setup as a **task library** for the running of **LISTLOG.**
    - So the library will be searched for any other programs that **LISTLOG** might decide to load.

If a task library is set up, then XDCCALL[A]/CMD[A] will **search that library first** for any module your program needs to load.

If a module is not found in the task library, the search will continue with the rest of the normal search order.

APF Authorization

XDCCALLA and **XDCCMDA** (for debugging **authorized** programs) have an added feature in their support for task libraries. They will cause the task library concatenation to be treated as being APF authorized regardless of whether or not the libraries within the concatenation actually are authorized.

This is of great convenience to the developer, but it also creates security concerns. However, the security issues can be controlled by the proper creation of Z/XDC related security rules. See **HELP SECURITY** for complete information.

For example, using **system Security** rules, you can restrict who is and isn't allowed to use the the APF authorization variations (XDCCALLA and XDCCMDA) of the XDCCALL load module.

You also can restrict who is allowed to load from the libraries containing the XDCCALLA and XDCCMDA load modules (again, using **System Security** rules).

Using **Z/XDC security rules,** you can restrict who is and is not allowed to run authorized debugging sessions. (See **HELP SECURITY AUTH.)**

Help XDCCAll Tasklib Tasklibraries

Task libraries work like JOBLIBs and STEPLIBs. They identify a concatenation of one or more libraries that the Operating System can search in response to a LOAD, LINK, ATTACH or XCTL macro. In fact, STEPLIBs and JOBLIBs are nothing more than task libraries that the Operating System establishes when it ATTACHs a job step's main program for execution.

When a task library is established for a subtask, that task library is available not only to that subtask but also to **all descendant tasks** of that subtask. In other words, when a given task or any one or more of its ancestor tasks has task libraries established, then **all** of those task libraries will be searched for LOADs, LINKs, ATTACHs, and XCTLs issued by that task, and task libraries established for more recent ancestors will be searched ahead of those established for older ancestors.

Support for task libraries is provided by certain programs, and the ddname of a task library will be whatever that supporting program decides it should be.

If a task library supporting program sees that the ddname is present, it will first OPEN that ddname and then provide the address of the OPEN'd DCB to an ATTACH macro via its TASKLIB= operand. The subtask that is being ATTACH'd will then have that library available as a task library.

As mentioned above, **STEPLIB** and **JOBLIB** are task library ddnames that the Operating System recognizes. When those DD cards are present, z/OS will OPEN one or the other of them and establish it as a task library when it ATTACHs an address space's main program as its jobstep task.

ISPLLIB is an example of a task library that ISPF establishes for programs LOAD'd within ISPF.

TASKLIB is a task library ddname recognized by the XDCCALL (et al.) program. When it sees that the TASKLIB ddname is present, it OPENs that ddname and then makes it available as a task library for the program to be debugged.

XDCCALL also has a mechanism by which an arbitrary ddname can be set up as the target program's task library. For more information, return to reading **HELP XDCCALL TASKLIB.**

Help XDCCAll Jobsteptasks

Normally, when xxxCALL[A] is setting up a program for debugging, it ATTACH's that program as a normal subtask. However, some programs contain logic that requires that they run as **jobstep tasks.** Well, xxxCALLA can do that too.

If you need your program to be ATTACH'd as a jobstep task, then tell xxxCALLA by including a **//xxxJSTCB DD DUMMY** allocation in your batch job's JCL. When xxxCALLA sees this, it will ATTACH your program as a jobstep task.

If for some reason xxxCALLA is unable to ATTACH jobstep tasks, then it will issue a warning message **(DBC547W)** and then proceed by ATTACH'ing your program as a normal task.

Help XDCCAll Ddnames

Preliminary notes

The term **xxxCALL** refers to the XDCCALL program. The **xxx** denotes the fact that when Z/XDC is renamed to some three character name (referred to herein as **xxx)** other than **XDC,** then XDCCALL must also be similarly renamed.

The xxxCALL program has three alias names: xxxCALLA, xxxCMD and xxxCMDA. The name **xxxCALL** will be used herein to collectively refer to all four names.

For more information, see HELP XDCCALL.

DDNAMEs Recognized by xxxCALL

The following ddnames are all optional. If xxxCALL finds them, then it will perform the indicated actions. (Note, if XDC has been renamed for example, to something such as **xxx,** then all ddnames that start with XDC will have to be changed to start with whatever matches **xxx.** See HELP XDCCLONES for more information.)

Note, the following are only those ddnames that are recognized and supported by the xxxCALL program. For a complete list of all ddnames recognized across the entire Z/XDC product, see HELP DDNAMES.

//TASKLIB DD DSN=loadlibraries
//STEPLIB DD DSN=loadlibraries
You can use either or both of these to point to load libraries that your debugging session is going to need. Generally, it doesn't matter which you you use, but you would use both if you're debugging an authorized program (via **XDCCALLA)** and your program needs some modules from authorized libraries (use **//TASKLIB)** while others need to come from non-authorized libraries (use **//STEPLIB). For more information, see HELP XDCCALL TASKLIB.**

//xxxCMDS DD DSN=sequentialfile
This must point either to a sequential file or to a specific member of a partitioned dataset. The file should contain Z/XDC commands. xxxCALL will cause Z/XDC, when it first gains control, to read this file and execute the Z/XDC commands that it finds there. The file may have any valid RECFM, LRECL, and BLKSIZE attributes. The records may (or may not) contain 8-digit sequence numbers located either in the last eight bytes (for RECFM=F files) or in the first eight data bytes (for RECFM=V or =U files).

//xxxFPTnn DD DUMMY
This ddname is deprecated. For the time being, it continues to function as described here, but it no longer is needed, because now FRR support will create Proxy Tasks as needed; they no longer need to be pre-created.

This ddname gives the number of FRR Proxy Tasks ("FPT") xxxCALL[A] is supposed to pre-create. **nn** must be a 2-digit decimal number specifying the number of tasks. Its value may range from 00 to 99. When this ddname is omitted, xxxCALL, by default, will not create any Proxy Tasks.

When Z/XDC is running as an FRR (for debugging SRBs and FRR protected tasks), it must use a "proxy task" for doing such things as communicating with the user and processing his commands. For more information, see HELP DEBUGGING FRR.

//xxxJSTCB DD DUMMY
If this ddname is present, then it needs to be a dummy allocation. When present, this ddname signals xxxCALLA that it should ATTACH the target program as a **jobstep task.** When absent, the program is ATTACH'd as a normal task. For more information, see HELP XDCCALL JOBSTEPTASKS.

//xxxLIB DD DSN=loadlibraries
This points to a library containing a particular copy of Z/XDC to be executed. (Normally, Z/XDC would be executed from task-, STEP-, JOB- or system libraries.) An xxxLIB file can be used to provide special copies of the xxx, xxxTFS, and xxxHELPM load modules. It can also provide special copies of the xxxUCI and xxxXITS exit modules if the local installation has written them.
(xxxLIB **cannot** be used for special copies of xxxCALL itself or its aliases. Use STEPLIB or JOBLIB instead.)

//XDCPROF DD DSN=profilelibrary
//ISPPROF DD DSN=profilelibrary
You can use either or both of these to point to FB-80 libraries for storing your Session Profiles. When both are present:
- //XDCPROF is read first for profile reads.
- //XDCPROF is is used for all profile saves.

For more information specifically about this, see HELP PROFILES DDNAMES.

If your TSO session allocates its //ISPPROF file with **/DISP=SHR,** then that same library can be used here, and then changes made here will be seen when you use Z/XDC in TSO.

When you do use the same library, corruption of library members is "possible" when both the TSO session and Z/XDC in the background job decide at the same time either to add new profile members or expand existing members. However, the likelihood of such corruption is mitigated by the fact that usually updates to a profile library from two different sources (Z/XDC and ISPF, for example) occur as the result of actions by a single human being, and it is rare for a human to do two things exactly at once.

If your TSO session allocates the ISPF profile library with DISP=OLD, then your

background job should point to any suitable FB-80 library that you want to store Z/XDC profiles into.

//xxxQUICK DD DUMMY

If this ddname is present, then it needs to be a dummy allocation. When present, this ddname signals xxxCALL that it should **not** trap execution to Z/XDC before starting the user's program. Instead, the user's program should just be allowed to start up immediately.

Note, Z/XDC's Startup Panel in ISPF has a **Quick Start?** field by which the user can request this quick start processing. For more information, see HELP DEBUGGING ISPF PANEL.

//xxxREFR8 DD DUMMY

This is a keyword ddname. Its purpose is to notify xxxCALL that the load modules for the program being debugged must be loaded into **key 8** storage even if they are **refreshable.** (xxxCALL accomplishes this by causing TCB_REFRPROT_OVERRIDE to be set on prior to the start of the debugging session.) For more information, see HELP DEBUGGING REFRESHABLE.

//xxxRENT8 DD DUMMY

This is a keyword ddname. Its purpose is to notify xxxCALL that the load modules for the program being debugged must be loaded into **key 8** storage even if they are **reentrant** and even if they reside in APF libraries. (xxxCALL accomplishes this by causing TCBTCP to be set on prior to the start of the debugging session.) For more information, see HELP DEBUGGING REENTRANT.

Help XDCCLones

Z/XDC contains support for being renamed from XDC to any other legal 3-character name. Such a renamed Z/XDC is referred to as an **XDC clone,** and the chosen name is referred to as a **clone name.** In Z/XDC documentation, clone names are usually represented by **xxx.**

The primary reason one would want to create an XDC clone is to allow the coexistence of multiple releases of Z/XDC on one system. For example, when installing a new release of Z/XDC, the Systems Programmer might want to temporarily install it using a different name so that the user community can have the opportunity to test out the new release without having to abandon the old release first.

When a Systems Programmer does decide to create an XDC clone, he first has to choose a legal 3-character name. This might be the release name (V31, for example), his own initials, or anything else he might care to choose.

Then the Systems Programmer has to rename all of Z/XDC's load modules. The modules are located in the distribution libraries whose factory default names are:

hlq.XDCV31.XDCLINK
hlq.XDCV31.XDCLINKE

hlq.XDCV31.XDCPLPA
The names of all of Z/XDC's load modules that matter start with the characters **XDC.** To create the clone, the Systems Programmer has to rename all of those modules so that they start with the chosen clone name instead of with "XDC". The best way to do this is to use ISPF's Library Utility (option =3.1).

Note, whenever we publish a beta release of a new Z/XDC, we will ship it already renamed to a suitable clone name. This is, of course, so that customers may try out a beta prior to committing to it. The clone name makes it possible for the beta release to coexist with any other Z/XDC release.

Quick Locate Support

Included in the Z/XDC product is a macro named **#XDCLOC8** which, when the **XDC** load module resides in common storage (the PLPA or DLPA), can be used to obtain the address of the XDC load module without having to use z/OS's LOAD services.

Well, the #XDCLOC8 macro can provide that same service for one clone **if** (and only if) the following conditions are met:
- The clone's root load module (the **xxx** load module) is located in common storage (on the DLPA or in the PLPA),
- The clone (or its Server task) has run one time authorized (so as to allow it to install its System Interface),
- This clone is the **first** clone to install its System Interface following an IPL. [THERE CAN BE ONLY ONE!]

The status of Quick Locate support can be displayed by the **LIST XDC** command.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

USING - Using XDC clones: Considerations and Examples.
AXDCIS - Avoiding hard-coding Z/XDC's name (for Assembler programmers).
CXDCIS - Avoiding hard-coding Z/XDC's name (for C programmers).
PROFILES - Sharing session profiles across clones.

Help XDCCLones Using

Generally, when using an XDC clone, in just about every instance where you would have used the name "XDC", you instead have to use the clone name. This includes program names, ddnames, VTAM names, ENQ names, system interface names, etc.

The major exception is security rules. All Z/XDC related security rules, including those that include "XDC" as a part of their names, remain unchanged. All XDC clones will use "XDC" (not their clone names) when building rule strings for security checks. (For more information, see HELP SECURITY RULES.)

Suppose that you want to use an XDC clone that has been named **V31.** Then the following are specific examples of changes that would have to be made in order to use that clone.

xxxCALL/xxxCALLA

Rather than invoking XDCCALL, you would instead invoke V31CALL. Example:

```
    //DEBUG   EXEC PGM=V31CALLA,PARM=IEFBR14
```

DDNAMES

DDnames used by a V31 clone of Z/XDC would change as follows:

- **//XDCCDF DD DUMMY** would become **//V31CDF DD DUMMY**
- **//XDCCMDS DD DSN=...** would become **//V31CMDS DD DSN=...**
- **//XDCJSTEP DD DUMMY** would become **//V31JSTEP DD DUMMY**
- etc.

For a complete list of ddnames applicable to various part of Z/XDC, see HELP DDNAMES.

Examples:

```
    //DEBUG   EXEC PGM=V31CALL,PARM=IEFBR14
    //V31PROF  DD  DISP=SHR,DSN=DBCOLE.ISPF.ISPPROF
    //V31RENT8 DD  DUMMY
    //V31REFR8 DD  DUMMY
    //V31QUICK DD  DUMMY
    //V31TRACE DD  SYSOUT=*
```

ISPF

When using Z/XDC's Startup Panel in ISPF, the panel has a fill-in field labeled **XDC's name ===>.** You will have to change this field's value to V31. When you do, you will see the panel automatically adjust to invoke Z/XDC via the "V31" clone name instead of via "XDC". Further, the change will persist from session to session until you decide to change it again. For more information, see HELP DEBUGGING ISPF PANEL.

Avoiding Hardcoding Z/XDC's Name in Assembler Programs

If your Assembler program uses a **LOAD** macro to locate the XDC load module, you can use either the **#XDCIS** macro or the **AXDCIS** function subroutine to make your program independent of Z/XDC's current name. You would use this macro or function in conjunction an **//XDCISxxx DD DUMMY** allocation. The **xxx** portion of the ddname is extracted and made available to your program to use as Z/XDC's current name. For more information about this **//XDCISxxx** ddname, see HELP DDNAMES XDCIS.

For more information about the **#XDCIS** macro and the **AXDCIS** function subroutine, see HELP XDCCLONES AXDCIS.

Avoiding Hardcoding Z/XDC's Name in C Programs

If your C program has a reason to know the clone name for the Z/XDC that is going to be used for debugging, then you can invoke the **CXDCIS** function subroutine to find out what that name is. You would use this function in conjunction an **//XDCISxxx DD DUMMY** allocation. The **xxx** portion of the ddname is extracted and returned as the function's value. For more information about the **CXDCIS** function subroutine, see HELP XDCCLONES CXDCIS.

#XDCHOOK Macro

If your program uses the **#XDCHOOK** macro, it will have to be changed to use the **XDC=** operand to identify the Z/XDC clone that is to be used. Examples:

```
    #XDCHOOK XDC='V31'
*or*
    L     R5,=CL4'V31'
    #XDCHOOK XDC=(R5)

*or* [This is the best choice]
    #XDCIS   ANSWER=XDCNAME
    #XDCHOOK XDC=XDCNAME
    ...
    XDCNAME DS CL8
```

For more information, see HELP HOOKS STATIC #XDCHOOK.

JES2

If you are using the JES2 interface routines that we distribute for debugging JES2 code, then you will have to add a **//XDCISxxx DD DUMMY** card to JES2's startup JCL to notify the interface code about which XDC clone is to be used. For more information, see HELP DEBUGGING JES2.

CICS

If you are using the XDC transaction (XDCCICSX load module) for using Z/XDC within CICS, then you will have to add a **//XDCISxxx DD DUMMY** card to CICS's startup JCL to notify the transaction about which XDC clone is to be used. For more information, see HELP DEBUGGING CICS.

VTAM Connections to cdf/XDC

If you want to debug a program running in the batch and you want to connect to the debugging session from an idle VTAM terminal, then the command you will have to issue will be (for example) **LOGON APPLID=V31.** (See HELP XDCSRVER CDF LOGON.)

Help XDCCLones Axdcis

The audience for the following discussion is Assembler programmers. If you program in C, then you should read HELP XDCCLONES CXDCIS.

A commonly used method for setting up a Z/XDC debugging session is to issue a LOAD macro for the load module named XDC and then:

- Either issue an ESTAE/X to make Z/XDC your program's ABEND recovery routine,
- Or issue a BASR to the XDC module from your existing ESTAE/X routine.

An alternative (and even better) method is to use the **#XDCHOOK** macro to both establish a Z/XDC debugging session and to pass control to it for initial commands.

These (and other) methods are fine; however, if you want to create the flexibility of being able to use Z/XDC clones, then you need to code some method by which you can provide Z/XDC's current name at execution time instead of at assembly time. Several possibilities come to mind...

- **Create a Command**
  If your program already has a command interface of some sort, then you might consider adding support for something like an **XDCIS=** command by which you can notify your program about the name to use for Z/XDC. (The default, of course, should be "XDCIS=XDC".)

- **Use the PARM Field**
  If your program already has code to parse and accept parameters received via the JCL PARM field, then just add support for an **XDCIS=** (or some such) PARM.

- **Keyword ddname [This is my favorite.]**
  A somewhat simpler way is to scan the TASK I/O Table (TIOT) for what I call a "keyword ddname". These are ddnames that are never OPEN'd, but whose presence or absence is scanned for in order to cause a program to take or not take some action.

  Your code could look for an allocation whose ddname starts with **XDCIS** and then use the next 3 characters as the XDC clone name. Then all you would have to do is add a dummy DD card to your JCL that looked something like this: **//XDCISxxx DD DUMMY.** Then if your program found this allocation, it would know that Z/XDC's clone is named whatever **xxx** is (instead of "XDC").

The #XDCIS macro

As a matter of fact, the Z/XDC product includes a macro that does exactly that - it scans the TIOT for a //XDCISxxx allocation.

The macro's name is **#XDCIS,** and what is does is, it generates a short piece of code that scans the current (or given) Task I/O Table (TIOT) looking for an //XDCISxxx ddname, and it returns the **xxx** value for use by a LOAD EPLOC= macro.

If no //XDCISxxx allocation is found, then the #XDCIS macro returns Z/XDC's default

name: **CL8'XDC'**

The **#XDCIS** macro returns the 3-character clone name in an 8-byte wide field, and it clears the remaining 5 bytes to blanks so the name is ready to be used directly by a **LOAD** macro. Here is an example of using #XDCIS...

```
            #XDCIS ANSWER=XDCNAME
            LOAD   EPLOC=XDCNAME
            ...
    XDCNAME DS     CL8
```

So if the Z/XDC clone that you need to use is named V31, then the DD card that you would add would be:

```
    //XDCISV31 DD DUMMY
```

For more information about the //XDCISxxx ddname, see HELP DDNAMES XDCIS.

The #XDCIS macro can be found in the hlq.XDCV31.XDCMACS library. For complete usage information, read the commentary located within the macro.

The AXDCIS Function Subroutine

The Z/XDC product also includes a function subroutine that does pretty much the same thing as the #XDCIS macro. It scans the TIOT for a //XDCISxxx allocation.

The function's name is **AXDCIS,** and what is does is, it scans the current Task I/O Table (TIOT) looking for a //XDCISxxx ddname, and it returns with R15 pointing to an 8-byte buffer containing the **xxx** value, left-aligned and right-padded with blanks. So the xxx value is ready for use by a LOAD EPLOC= macro. Here is a usage example:

```
            L     R15,=V(AXDCIS)
            BASR  R14,R15
            LOAD  EPLOC=(R15)
```

So if the Z/XDC clone that you need to use is named V31, then the DD card that you would add would be:

```
    //XDCISV31 DD DUMMY
```

For more information about the //XDCISxxx ddname, see HELP DDNAMES XDCIS.

If no //XDCISxxx allocation is found, then the AXDCIS function returns Z/XDC's default name: **CL8'XDC'**

The AXDCIS subroutine can be found in the hlq.XDCV31.XDCLINK library. It either can be linked into your program's load module, or it can be LOAD'd and called.

The source code for AXDCIS can be found in hlq.XDCV31.XDCASM(SRCAXDCI).

Help XDCCLones Cxdcis

The audience for the following discussion is C programmers. If you program in Assembler, then you should read HELP XDCCLONES AXDCIS.

A commonly used programming method for referring to the Z/XDC product is via it's primary name: **"XDC".** In fact, Z/XDC's primary load module is named **XDC.** Well, that's fine as far as it goes, **but** if you want to maintain the flexibility of being able to use XDC clones (such as **beta releases** of Z/XDC), then you need to code some method by which you can provide Z/XDC's current name at execution time instead of at compile time. Several possibilities come to mind.

- **Create a Command**
  If your program already has a command interface of some sort, then you might consider adding support for something like an **XDCIS=** command by which you can notify your program about the name to use for Z/XDC. (The default, of course, should be "XDCIS=XDC".)

- **Use the PARM Field**
  If your program already has code to parse and accept parameters received via the JCL PARM field, then just add support for an **XDCIS=** (or some such) PARM.

- **Keyword ddname [This is my favorite.]**
  A somewhat simpler way is to scan the TASK I/O Table (TIOT) for what I call a "keyword ddname". These are ddnames that are never OPEN'd, but whose presence or absence is scanned for in order to cause a program to take or not take some action.

  The code could look for an allocation whose ddname starts with **XDCIS** and then use the next 3 characters as the XDC clone name. Then all you would have to do is add a dummy DD card to your JCL that looked something like this: **//XDCISxxx DD DUMMY.** Then if your program found this allocation, it would know that Z/XDC's clone is named whatever **xxx** is (instead of "XDC").

The CXDCIS Function Subroutine

Distributed with the Z/XDC product is a C function named **cxdcis()** that looks for a DUMMY file allocated to ddname **//XDCISxxx** (where **xxx** is the Z/XDC clone name that your program should use when referring to Z/XDC). If such a ddname is found, then the **xxx** part of the ddname is extracted and placed, blank-padded (and null-terminated), into an 8-character wide string. This string is then returned to the function's caller as the function's value.

On the other hand, if no such ddname is found, then **XDCbbbbb** (where "b" here represents a blank) is returned as the function's value.

Example: if you want your program to use a clone of Z/XDC named **V31,** then...

- Add to your runtime JCL the DD card **//XDCISV31 DD DUMMY**

- Add to your program the statement: **#include "cxdcis"** This includes a header file that defines the **cxdcis()** function and maps it to an Assembler routine named **AXDCIS** (not CXDCIS). AXDCIS is described in HELP XDCCLONES AXDCIS.

- Also add to your program a statement such as **char*xdcname = cxdcis()** (This calls the **cxdcis()** function and assigns the returned clone name to the xdcname variable.)

- Use the **xdcname** variable at all points in your program where you might reference Z/XDC.

- The **CXDCIS** header file is located in hlq.XDCV31.XDCSAMP, so you need to add LSEARCH(//'hlq.XDCV31.XDCSAMP') to your compile time options file.

- The **AXDCIS** function subroutine load module is located in hlq.XDCV31.XDCLINK, so you need to include that library into your bind-time //SYSLIB concatenation.

An example of using the **cxdcis()** function can be found in a sample program named **XDCCSAMP:**
- The source code for XDCCSAMP can be found in hlq.XDCV31.XDCSAMP(SRCCSAMP).
- Sample JCL for compiling XDCCSAMP can be found in hlq.XDCV31.XDCJCL(CMPCSAMP).
- Sample JCL for binding XDCCSAMP can be found in hlq.XDCV31.XDCJCL(LCSAMP).

For more information about c/XDC, see HELP DEBUGGING C.

Help XDCCLones Profiles

An XDC clone will create its own session profiles for use during the debugging session. If, however, you want to load a profile from a different XDC (or from a different clone of XDC), you can do so by specifying the clone name as the PROFILE READ command's second operand. For additional information, see HELP PROFILES READSAVE.

Examples:

PROFILE READ
PROFILE READ XDC
PROFILE READ XDC V31
(Continuing under the assumption that you are using an XDC clone named V31) All three of these commands do the identical thing: They attempt to read your personal default profile from the XDC clone named V31.

PROFILE READ XDC DAV
This reads your personal default profile from an XDC clone named DAV.

PROFILE READ XDC XDC
This reads your personal default profile from XDC.

PROFILE READ TEMP FHC
This reads a saved profile named TEMP from an XDC clone named FHC.

PROFILE READ XDC TFS
This reads your Data Center's default profile.

PROFILE RESET AWIDE
This initializes your profile data from factory defaults that have been assembled into the V31 load modules and that are suitable for Assembler programmers using large geometry terminals.

PROFILE RESET C80
This initializes your profile data from factory defaults that have been assembled into the V31 load modules and that are suitable for C programmers using small geometry (80-column wide) terminals.

PROFILE SAVE
This saves your session's current profile data as your personal default for the XDC clone that you are currently using (V31 in this example).

Help XDCSrver

server/XDC is a server address space that is needed in support of some of Z/XDC's commands and capabilities. Basically, server/XDC is a permanent, non-swappable address space that hosts one or more internal **subservers,** each of which supplies a service that is needed by one or another of Z/XDC's commands and capabilities.

When server/XDC is down, Z/XDC debugging sessions are not interrupted, but those commands that require server services will not be usable until the server is brought back up.

Also, the caption **(DBC496W!)** is inserted into the title line (at the left) for all debugging sessions. See **HELP MESSAGES DBC496** for a detailed discussion of what commands and services cannot be used when server/XDC is down.

Sometimes, it becomes necessary to bounce server/XDC. Typically, this can occur when issues arise with the VTAM node and APPLIDs that the cdf/XDC subserver uses. So the question arises... Is it safe to bounce server/XDC? Generally, yes. See **HELP XDCSRVER ISBOUNCINGSAFE?** for more information.

Currently, the following Z/XDC commands and services require the server/XDC to be running.

Batch Program Debugging

When you want to debug a program running in a non-TSO address space (such as a Batch Job or a System Task), a mechanism is required by which an interactive user terminal can be connected to the debugging session so that you, the user, can issue Z/XDC commands. That mechanism is called **cdf/XDC.** cdf/XDC runs as a subserver within server/XDC. When that subserver is not available, logons to pending debugging

sessions cannot occur.

Existing sessions, however, are not affected. server/XDC has no involvement with ongoing sessions. For more information, see **HELP XDCSRVER CDF.**

c/XDC Support

Some of our c/XDC support is written in Assembler, but a lot of it is written in XL C/C++. In order to facilitate c/XDC's need to not interfere with the user program's LE requirements, it is necessary for c/XDC's LE code to be located in an entirely different address space. To accomplish this, we wrote a c/XDC subserver that runs within server/XDC. This subserver is normally automatically started when the server/XDC is started. However, if, for some reason, the c/XDC subserver is not available then c/XDC cannot be used.

For more information, see **HELP DEBUGGING C.**

HOOK Command Support

For the most part, Z/XDC's Hook Support functions without the need for external support routines, but under certain limited conditions, a PC routine is needed for hook removal. Now, unlike SVC routines, PC routines generally need to be owned by a server space, and when that server space is not present, the PC routine is not present either.

In most situations when a hook needs to be removed, Z/XDC just does so with a simple MVC. But when the following circumstances occur, a PC routine must be called to do the removal:

- While running authorized, Z/XDC's **HOOK** command was used to set a hook into a program located in key 0 storage.
- But when program execution reached the hook, it was running in problem state and unauthorized.
- Since Z/XDC's Hook Support runs in the target program's execution environment, it does not have the authority to remove the hook.
- So a PC routine must be called to perform the removal.

If server/XDC is not running, then the necessary PC routine does not exist, so in the above scenario, Hook Support is unable to remove the hook, and so it has no recourse but to issue a message **(DBC548T)** and then ABEND the program (at a DEAD trap).

Bottom Line

If you want to:
- Logon to a background debugging session,
- Use c/XDC,
- Use the HOOK command under the circumstances described above,

Then you must ensure that the server/XDC address space is up and running.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

ISBOUNCINGSAFE? - Generally yes. But read this for details, how-tos and considerations.
CDF - About logging onto a debugging session that is pending in a background job or system task.

Help XDCSrver Isbouncingsafe?

cdf/XDC uses VTAM to communicate with 3270 workstations. It happens more frequently than we'd like that VTAM connection issues arise that are often best solved by bouncing the server/XDC address space.

Such issues can manifest themselves as hangs when users attempt to connect to their debugging sessions. They also can lead to unexpected error messaging from Z/XDC pertaining to VTAM nodes, APPLIDs, RPLs and the like.

There are multiple reasons why VTAM nodes can go south, many of which arise externally. So rather than messing around trying to diagnose a problem, perhaps the first thing to try is to bounce both server/XDC and the VTAM node that cdf/XDC uses. It is both a safe and easy thing to do.

FAQs

- **Is it safe?** Yes. Bouncing server/XDC solves problems. It doesn't create them.

- **Is it disruptive?** That depends...
    - If you intend to bounce just server/XDC, then disruption is minimal and possibly won't even be noticed by the users.
    - If you intend also to bounce the VTAM node that cdf/XDC uses, then active users will be disconnected from their sessions; however, the session does not end. Instead, the users will be able to reconnect once the node and the server are brought back up.

- **Is it difficult?** No. It takes just two operator commands to being the server and the node down, and two more to bring them back up.

- **Can bouncing fail?** Sometimes. If bouncing does not fix the problem, then there may be issues with the VTAM node and the APPLID definitions that it contains.

  There are two control files involved in the definition and creation of cdf/XDC's VTAM node:

  - server/XDC's startup JCL includes a **//SYSIN DD** file that contains parameters that define the names both of the major node used by cdf/XDC and the APPLIDs contained within that node. See commentary within the file for detailed information about the parameters.

  - The factory default version of the SYSIN file can be found in **hlq.XDCV31.XDCSRVER(XSRVPARM).**

  - The APPLID definitions themselves are found in the member of VTAM's **VTAMLST** library concatenation whose name is given by the **VBUILD** parameter. The factory default for that name is **XDCCDF.**

  - The factory default version of the VTAMLST member can be found in **hlq.XDCV31.XDCSRVER(XSRVVTAM).**

  The factory default versions of both the SYSIN file and the VTAM node definition can be found in the **hlq.XDCV31.XDCSRVER** library.

- **What happens when server/XDC is down?** When server/XDC is down, the services that it provides are unavailable to Z/XDC; however, those services are restored as soon as server/XDC comes back up. For a detailed discussion of the interrupted services, see **HELP MESSAGES DBC496.**

  In particular, ongoing debugging sessions are **not interrupted,** by server/XDC being down. In fact, unless a user attempts to issue one of the affected commands, he likely won't even notice the bounce, particularly if the bounce is brief. (However, see the following.)

- **What happens when cdf/XDC's VTAM node is down?** All users who are connected to their debugging sessions via cdf/XDC are abruptly disconnected. However, their debugging sessions remain, so the users can reconnect to them once the node is restarted.

  Users running debugging sessions within TSO are unaffected.

How to Bounce server/XDC and the VTAM Node Used by cdf/XDC

There are two things that you will have to bounce:
- The server itself.
- And the VTAM node that the server uses.

Here's what to do.

1) Navigate to a SYSLOG interface. Typically, if you're in ISPF, you would use **=M.5** to go to IBM's **SDSF.**

   Then you would use SDSF's **LOG** command to access SYSLOG. (If you do not have the security credentials to access SYSLOG, then you will have to ask someone else to do this.)

2) Issue **/C XDCSRVER** to bring down server/XDC. Notes:

   - server/XDC's jobname might, of course, be something different from XDCSRVER. In particular, older customers may still be using XDCCDF as the jobname.

   - The **/** is SDSF's command leader character. If you are using a different SYSLOG interface, you will use a different leader (or perhaps no leader at all).

   - It is recommended that you bring down server/XDC with a **CANCEL** command. server/XDC also accepts a **STOP** command **(P),** but if the server is particularly hosed, **P** might not work.

2a) If the **CANCEL** command fails to bring XDCSRVER down, you may have to issue **/FORCE XDCSRVER.**

   This would be a rare necessity. If it is needed, we would be interested in seeing a dump, so if you need to resort to FORCE, please obtain a dump first. See **HELP SUPPORT SENDINGUSDUMPS** for what we would need such a dump to include.

3) Issue **/V NET,INACT,I,ID=XDCCDF** to bring down the VTAM node that XDCSRVER uses. Note:

   - This command assumes that the VTAM node used by server/XDC is named **XDCCDF.** It may, of course, be named anything else.

4) Once everything settles, issue **/V NET,ACT,ID=XDCCDF** to restart the VTAM node.

5) Finally, issue **/S XDCSRVER** to start a new instance of server/XDC.

   The server will display its normal **System Interface Initialization Report** (messages DBC514I)...

   ...Eventually followed by its normal **DBC655I ENTER "F XDCSRVER,HELP" FOR A LIST OF VALID COMMANDS.**

   Notice: **The System Interface Initialization Report** should not report that

anything was installed, reinstalled or otherwise replaced. The expected STATUSes should be:
- ALREADY PRESENT in most cases.
- NOT PRESENT in one case.
- SET or NOT SET in two other cases.

Anything else indicates that a prior Maintenance Install Process has been unexpectedly left pending. This may lead to issues that may require a reIPL.

6) Retest for the problems that were being experienced. They should now be gone.

Help XDCSrver Cdf

Debugging Batch Jobs and System Tasks

cdf/XDC makes it possible for terminal users to debug background programs, i.e. batch jobs and system tasks. The background job may be running on any processor in a CF-connected Sysplex.

cdf/XDC is a communications program that provides a user with a full-featured, fullscreen terminal interface connected to a Z/XDC debugging session running in a background address space: a batch job or a "started task" (a system program started by an operator issued "START" command).

Using cdf/XDC in TSO

cdf/XDC can also be used for debugging programs running in **TSO** addresses spaces. Normally, when Z/XDC is used within ISPF or TSO, it will use fullscreen TPUTs/TGETs or ISPF Display Services to send its displays to the TSO user’s terminal (the same terminal that is being used by TSO and by ISPF for communicating with the user). However, when Z/XDC is being used to debug a TSO or ISPF program that itself issues fullscreen displays, that program’s fullscreen display management and Z/XDC’s fullscreen management will interfere with each other, with each program corrupting the other program’s displays. When this happens, the best solution is to cause Z/XDC to use an entirely different terminal for its displays. That can be done via cdf/XDC.

The cdf/XDC subserver

cdf/XDC is implemented through VTAM. In order to use it, your computer center must be running a system task named **XDCSRVER,** and the **cdf/XDC subserver** must be activated within XDCSRVER.

The cdf/XDC subserver functions as an interface between background jobs that have ABENDed or breakpointed to Z/XDC, and users that wish to debug those jobs from their terminals.

Setting Up a Batch program to use Z/XDC

In most cases, in order to use cdf/XDC to debug your batch job or system task, all you have to do is have Z/XDC established as your program's **newest** ABEND error recovery routine (see HELP DEBUGGING GETTINGSTARTED) at the time that your program either ABENDs or reaches a breakpoint. This causes Z/XDC to receive control when an ABEND or breakpoint occurs.

When Z/XDC receives control in the batch, its default action is to connect to cdf/XDC (running in the **server/XDC** address space) and to wait for a programmer to signon to the debugging session.

There are many ways to establish Z/XDC as your program's ABEND error recovery routine. Here are some of the more typical ways:

- Run your program under the control of the XDCCALL program. Briefly, XDCCALL attaches your program as a subtask and makes Z/XDC the ESTAI routine for your program and all of its subtasks (if any). See HELP XDCSRVER CDF SETUP for more information.

  If your program sets up its own ABEND recovery (ESTAE[X] macros, ARRs, ATTACH with ESTAI=, etc.), then those recovery routines will interfere with Z/XDC. The easiest way to cope with such interference is to use Z/XDC's **HOOK** and/or **HDEFERRED** commands to reestablish Z/XDC as the newest recovery routine in the locally protected code. See HELP HOOKS fore more information.

- Modify your program in either of two ways:
  - LOAD Z/XDC into storage and issue an ESTAE macro making Z/XDC your newest ESTAE.
  - Or modify your existing ESTAE to BASR to Z/XDC.

  Depending upon circumstances, you may **both** have to use XDCCALL **and** have to modify your program. See the following for additional information:
  HELP DEBUGGING ESTAES
  HELP DEBUGGING EXITROUTINES

Using HOOK to start Debugging On the Fly

If your background program is already running, then you can (if system security permits) dynamically "hook" an Z/XDC debugging session into it. Proceed as follows:

- Log onto a TSO session.
- Either from TSO's **READY** prompt or from ISPF's **=6** option, type **XDCCALLA IEFBR14** (or any other program you want) to start an authorized debugging session.
- Use Z/XDC's **SET ASID jobname** command to gain access to your background job.
- Use various **LIST, DISPLAY** and **FORMAT** commands to figure out where you want a debugging session to start.
- Use the **HOOK** command to insert at that location special code that will create a debugging session in your background job.

When program execution reaches your hook, a debugging session will be created in that address space, and you can then connect to cdf/XDC and log onto that session.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

↕	**SETUP**	- Setting up a background job or system task for interactive debugging. (The easy way is to use HOOK.)
↕	**COMMUNICATIONS**	- Controlling how a debugging session communicates with the programmer: via cdf/XDC, via WTO/WTORs, or via TSO.
↕	**CONNECTION**	- Managing your connection of the cdf/XDC session (attentions and disconnecting).
↕	**TSO**	- Using cdf/XDC for debugging a TSO or ISPF program. (Why and how)
↕	**LOGON**	- Connecting your terminal to a background debugging session.
↕	**COMMANDS**	- Z/XDC commands supported by or affected by the cdf/XDC environment.
↕	**FASM**	- A comparison of cdf/XDC with Z/XDC's Foreign Address Space Mode.
↕	**RECOVERY**	- Suggestions for recovering failed cdf/XDC connections.
↕	**CROSSSYSPLEX**	- Brief remarks regarding what is needed to install cdf/XDC with Cross Sysplex Support activated.

Help XDCSrver Cdf Setup

Strictly speaking, cdf/XDC cannot be used to connect a debugging session to just any arbitrary address space. The target background job must be specifically prepared to accept a cdf/XDC connection. In other words, the target batch job or system task must cooperate. You must take steps to make Z/XDC its newest ABEND error recovery routine (ESTAE, ESTAI, etc.).

When Z/XDC is your program's newest recovery routine, it will receive control whenever your program ABENDs or reaches a breakpoint. When Z/XDC receives control, it starts looking around for a way to connect to a terminal so that it can conduct an interactive debugging session with a user.

When Z/XDC finds that it is running within a TSO address space, its default action is to connect to the space's TSO terminal. However, that action can be overridden to cause Z/XDC to connect to cdf/XDC instead.

When Z/XDC finds that it is running within a batch job or system task address space, its default action is to try to connect to cdf/XDC.

Setting Up a Program to Use Z/XDC

Generally, there are four common ways to do this:

↕ - Use the **XDCCALL(A)** utility to run your entire program within a Z/XDC debugging session environment.
- Add code to your program to issue an ESTAE(X) to setup Z/XDC as your program's ABEND error recovery routine.
↕ - Add a **#XDCHOOK** macro to your code to setup Z/XDC as that code's ABEND error recovery routine.
↘ - If your program is already running, then you can start a debugging session

session **on the fly** by starting up Z/XDC in another address space and then using the **HOOK** command targeting the specific routine that you want to debug.

In practice, you will probably want to do a combination of these things.

Anyway, once Z/XDC receives control in a batch job or system task address space, connecting to cdf/XDC automatically happens. But if it receives control in a TSO address space, and you want it to use cdf/XDC instead of the TSO terminal, you will have to force it to make that choice. See HELP XDCSRVER CDF TSO for details.

Using XDCCALL(A)

XDCCALL and XDCCALLA are utilities that invoke your program as a subtask and establish Z/XDC as an ESTAI-type ABEND error recovery routine for your program. They also provide automatic support for such additional facilities as task library support, a user written commands exit routine, an initial commands string, etc. For more information, see HELP XDCCALL.

Use XDCCALL if you want to debug a non-APF authorized program. Use XDCCALLA if you want to debug an APF-authorized program. (In all other respects, XDCCALL and XDCCALLA are identical to each other.)

To run your program under XDCCALL or XDCCALLA, change your JCL as follows:

- Change your job's EXEC card to invoke the XDCCALL or XDCCALLA program instead of your normal program. At the same time, notify XDCCALL (or XDCCALLA) of your program's name by adding it to the start of the PARM field. Examples:

```
    //A EXEC PGM=MYPGM,...    becomes:
    //A EXEC PGM=XDCCALL,PARM='MYPGM'

    //A EXEC PGM=MYPGM,PARM=',,PARM1,PARM2'...   becomes:
    //A EXEC PGM=XDCCALL,PARM='MYPGM ,,PARM1,PARM2'...
```

  (XDCCALL and XDCCALLA will remove your program's name, and a trailing blank if any, from the PARM field before passing it on to your program.)

For more information, see:
    HELP XDCCALL
    HELP DEBUGGING BATCHJOBS

Making Z/XDC Your Program's ESTAE(X)

There are two ways to do this. One is by issuing a LOAD macro and an ESTAE(X) macro, first to find the XDC load module's address in storage (XDC will normally be located in the PLPA), then second, to make Z/XDC your program's ESTAE or ESTAEX routine.

The other way is to issue a #XDCHOOK macro to do essentially the same thing.

Once Z/XDC is set up, then you still need to pass control to it so that you can load your maps, set your breakpoints and such. For the LOAD/ESTAE(X) case, this can be done by coding a #DIE macro following the ESTAE(X) macro. (For the #XDCHOOK case, this is unnecessary, since the routine that received control from #XDCHOOK has a #DIE embedded within it.) For more information about #DIE, see HELP BREAKPOINTS DEADTRAPS #DIE.

Here is an example of using LOAD/ESTAE/#DIE for starting a debugging session:

```
...
LOAD  EPLOC==CL8'XDC',ERRET=NOXDC
ESTAE (R0)
#DIE  'HI. WHAT DO YOU WANT?'
...
```

See HELP DEBUGGING EXITROUTINES for discussions and more examples of this method.

When Things Get Complicated

In practice, for structurally complex programs, using XDCCALL(A) is not enough for setting up a successful debugging session. This is because XDCCALL(A) only sets up Z/XDC has your program's ESTAI. If your program then goes and issues ESTAEs, ESTAEXs, and ATTACHS-with-ESTAIs of its own, those newer recovery routines will interfere with Z/XDC. Also, if your program has modules, exit routines, and interrupt routines that run under their own Request Blocks, this too complicates the use of Z/XDC.

Briefly, to deal with these kinds of complications, you need either to integrate a Z/XDC interface into your own recovery routines, or you need to issue additional ESTAEs ESTAEXs, or HOOKs in order to ensure that, at any code point of interest, Z/XDC is the newest recovery routine in the environment. For more detailed information, see:

- HELP DEBUGGING ESTAES
- HELP HOOKS
- HELP DEBUGGING EXITROUTINES

Using HOOK to Start a Debugging Session on the Fly

Although it is generally required that an address space be intentionally prepared in order to debug it via cdf/XDC, there is a very easy way to force an unprepared address space into a debugging session. That's by using the **HOOK** command. All you have to do is:

- Logon to TSO, then start an authorized Z/XDC debugging session against any program. (IEFBR14 will do.) Example: Go to ISPF's option 6, and type **XDCCALLA IEFBR14.**

- Gain "Foreign Address Space Mode" access to the address space that you want to debug. Example: "SET ASID JES2".

- Locate the code that you want to debug in that address space, and use the HOOK command to set a Z/XDC trap there.

  Warning, if your target code is executed by multiple tasks, then the hook will trigger a debugging session in the first task (and **only** the first task) that executes the hooked code. This may or may not be the task that you want to debug, so you will have to exercise some careful planning and forethought when choosing what you want to hook. Tip: Set your hook into code that will be executed only by the task that you want to debug. Then once a debugging session has started and you've connected to that session, then use Z/XDC's normal breakpointing commands (AT, TRAP, etc.) to set traps into the actual code of interest.

- When the hooked code is executed, a Z/XDC debugging session will be created. If

the targeted address space is not a TSO space, Z/XDC will connect to the Cross Domain Facility and wait there for you to sign on and begin your debugging session.

On the other hand, if the targeted address space is a TSO space (and if Z/XDC's default action has not been overridden), then Z/XDC simply begins the debugging session by displaying messages at that TSO user's terminal. (For information about forcing a TSO debugging session to use cdf/XDC instead of the TSO terminal, see HELP XDCSRVER CDF TSO.)

Using a Personal Profile

If you wish to use your own personal profile of debugging session characteristics, then you will have to add a DD card to the background job's JCL. The ddname can be either ISPPROF or xxxPROF (it does not matter which), and it should point either to your TSO session's ISPF profile library or to a copy of that library.
(The **xxx** in **xxxPROF** must match Z/XDC's current name, usually **XDC.)**

If your TSO session allocates the ISPF profile library with DISP=OLD, then your background job should point to a **copy** of the library (not the library itself). On the other hand, if your TSO session allocates the ISPF profile library with DISP=SHR, then your background job could point to the same library. In some ways, this is more convenient; however, corruption of library members is possible when both the TSO session and Z/XDC in the background job decide at the same time either to add new profile members or expand existing members. (However, the likelihood of such corruption is mitigated by the fact that usually updates to a profile library from two different sources (Z/XDC and ISPF, for example) occur as the result of actions by a single human being, and it is rare for a human to do two things exactly at once.)

The items profiled by Z/XDC include PF key definitions, terminal display window placements, terminal colors, and many more. If you omit the xxxPROF and ISPPROF DD cards, then Z/XDC will use its factory default settings for all profile values. For detailed descriptions of what settings are profiled, see HELP PROFILES MENU.

Example:

Suppose that you wanted to debug IBM's IEBCOPY utility. (I don't know why you would want to do this, but then this is just a setup example.) Without a Z/XDC interface, the job's JCL might appear as follows:

```
//COPY   EXEC PGM=IEBCOPY
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD DSN=DBCOLE.OLDLIB,DISP=SHR
//SYSUT2   DD DSN=DBCOLE.NEWLIB,DISP=OLD
```

To setup a Z/XDC interface to cdf/XDC, you might change the JCL to the following:

```
//COPY   EXEC PGM=XDCCALLA,PARM='IEBCOPY'
//XDCPROF  DD DSN=DBCOLE.ISPF.ISPPROF,DISP=SHR
//*
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD DSN=DBCOLE.OLDLIB,DISP=SHR
//SYSUT2   DD DSN=DBCOLE.NEWLIB,DISP=OLD
```

Help XDCSrver Cdf COMMUnications

Once Z/XDC receives control in a background job, it must find an interactive interface by which it can communicate with a user. From a non-TSO address space, there are two possible interfaces that Z/XDC supports:

- **cdf/XDC** for fullscreen interactive communication with a user's terminal.
- **WTO/WTOR** macros for linemode interactive communication with a System Operator's console.

When Z/XDC runs in a TSO address space, there is a third possible communications interface: the user's **TSO terminal.**

Z/XDC's **default** communications interface depends upon whether Z/XDC is running in a TSO address space or a background address space. If it is running in TSO, then Z/XDC will want to use the TSO terminal. On the other hand, if Z/XDC is running in a background address space, then it will want to use cdf/XDC. In both cases, if the desired communications interface is not available, and if the user has not indicated otherwise, then Z/XDC will terminate rather than attempt to use other possible interfaces. (When Z/XDC "terminates", it percolates the ABEND. This means that ABEND processing proceeds as if Z/XDC had never been called.)

The user can control the choices that Z/XDC makes when selecting a communications interface. This is done by allocating special ddnames as dummy files. When Z/XDC needs to decided which interface to use, it searches system control blocks (TIOT DD entries) to see if these special ddnames exist. Z/XDC does not actually OPEN or otherwise use these ddnames. It only checks to see whether or not they are present. In other words, these special ddnames only serve as a simple signaling mechanism to Z/XDC.

Z/XDC's interface selection process is affected by the presence or absence in various combinations of the following ddnames:

  //xxxCDF DD DUMMY (background JCL)
  ALLOC FILE(xxxCDF) DUMMY REUSE (in TSO)
- This causes Z/XDC to select the cdf/XDC communication interface even in situations where it would not otherwise do so. The only necessary requirement is that the cdf/XDC subserver be up and running within the xxxSRVER started task.

  //xxxWTOR DD DUMMY (background JCL)
  ALLOC FILE(xxxWTOR) DUMMY REUSE (in TSO)
- This makes it possible for Z/XDC to select the WTO/WTOR communications interface.

(Note, the **xxx** in the names shown above and below must match Z/XDC's current name, usually **XDC.)**

The xxxCDF and xxxWTOR ddnames can be present or absent in four combinations with the following results:

- **//xxxCDF omitted**
  //xxxWTOR omitted
  There are two distinct cases here:
  - If Z/XDC is running in a TSO address space, then Z/XDC communicates with the user via his TSO terminal. If the TSO terminal is not available (hard to imagine), then Z/XDC terminates and percolates the ABEND.

    - If Z/XDC is running in a background address space, then it attempts to communicate with the user via cdf/XDC. If cdf/XDC is not available, then Z/XDC terminates and percolates the ABEND.

- **//xxxCDF present**
  //xxxWTOR omitted
  Z/XDC attempts to communicate with the user via cdf/XDC **regardless** of whether Z/XDC is running in a TSO address space or in a background address space. If cdf/XDC is not available, then Z/XDC terminates and percolates the ABEND.

- **//xxxCDF omitted**
  //xxxWTOR present
  Regardless of whether Z/XDC is running in a TSO address space or a background address space, it attempts to communicate via WTO/WTORs issued to the System Operator consoles. If the operators deny Z/XDC permission to use their consoles, then Z/XDC terminates and percolates the ABEND.

- **//xxxCDF present**
  //xxxWTOR present
  Regardless of whether Z/XDC is running in a TSO address space or a background address space, it first attempts to communicate with the user via cdf/XDC. If cdf/XDC is not available, then it attempts to communicate via WTO/WTORs issued to the System Operator consoles. If the operators deny Z/XDC permission to use their consoles, then Z/XDC terminates and percolates the ABEND.

Help XDCSrver Cdf CONnection

Starting a debugging session via cdf/XDC

When a program has been properly setup to use Z/XDC for debugging (see **HELP DEBUGGING GETTINGSTARTED),** then when that program ABENDs or reaches a breakpoint, Z/XDC will receive control. One of the first things that Z/XDC will do is try to find a connection by which it can communicate with a user.

When Z/XDC decides that the user connection will be via cdf/XDC (see **HELP XDCSRVER CDF COMMUNICATIONS),** it generally has to wait until a user logs onto the debugging session (and that usually takes some amount of time, at least a few seconds, often much more).

So Z/XDC notifies the connection broker (the cdf/XDC subserver), that it (a debugging session) is waiting for a user, and it issues a WTOR **(DBC640Q)** to give the System Operators some controls over the pending debugging session. Then Z/XDC goes into a WAIT state.

Eventually, a user will log into cdf/XDC and then, via a **Job Selection Menu,** connect to the debugging session. (See **HELP XDCSRVER CDF LOGON** for information about connecting to a debugging session.) This will cause Z/XDC to wake up in the target address space and get the session rolling.

When cdf/XDC has been appropriately set up, and when the debugging session is running in a Coupling Facility connected Sysplex, the user may connect into the debugging session from any processor connected to the sysplex. He does not have to be connected to the same processor as the debugging session.

Note, once the user has established the connection to the debugging session, the user's terminal communicates **directly** with Z/XDC running in the target address space. The cdf/XDC subserver (the connection broker) is no longer involved. In fact, even if the cdf/XDC subserver terminates, that will have no effect upon existing debugging sessions. They will simply continue uninterrupted.

Keyboard Lockouts

During your debugging session, there will be times when Z/XDC is in control, and there will be times when you have issued the **GO** command to give control back to your program.

When Z/XDC is in control, it usually will be awaiting commands from you, and so your terminal's keyboard will be unlocked. On the other hand, whenever your program is in control nobody will be awaiting commands, and so your keyboard will be locked. So if your program runs a long time without reABENDing or hitting another breakpoint, your keyboard will remain locked for that same long time. Patience may be needed.

The ATTN Key

However, if patience should run out, you can always press the **ATTN** key (which usually is mapped to the **ESC** key by most 3270 workstation emulation programs). What happens next, though, depends upon how your terminal is connected to the debugging program.

- **VTAM Connections**
  If you have connected to your debugging session via a direct logon to cdf/XDC **(LOGON APPLID=xxx),** then pressing the **ATTN** key will send an attention interrupt to Z/XDC, still lurking in you program's address space. This will cause Z/XDC's Interruption Handler to receive control and display **DBC893I** along with about **six paragraphs(!)** explaining what the current environment is and how you can proceed back to a normal environment.

  The information provided is both detailed and **customized** for the specific interruption environment. If you read the information and follow the instructions carefully, you should be able to find your way back to the interrupted routine.

  The **DBC893I** messages explain exactly how to do all that. See **HELP MESSAGES DBC893** for more information.

- **Connections from ISPF**
  If you have connected to your debugging session via a Z/XDC's Startup Panel in ISPF, then attentions are processed a bit differently. In this case, the attentions are **not** passed directly to the target program's address space. Instead, they are initially handled locally in your TSO session's address space.

  What the local attention handler does is display message DBC686Q offering several choices for what to do next. Those choices are:

  - **Forward the attention signal to the Batch Program Being Debugged**
    The program being debugged will be interrupted and issue a **DBC893I** message.

This creates the situation described above under **VTAM Connections.**

- **Disconnect the Terminal from the Debugging Session**
  This will free your terminal from the debugging session, allowing you to do other work in ISPF. Your background program will continue executing, unaffected in the disconnection. Be aware, however, that you will not be able to reconnect to the debugging session unless and until your background program reABENDs or again reaches a breakpoint, thereby allowing Z/XDC to receive control again in the address space. At that time, you will be able to reconnect.

- **Ignore Your Attention Signal and Resume Waiting**

  Your keyboard will be returned to its locked state, but your terminal will remain connected to the debugging session. You can, of course, press **ATTN** again to redisplay the **DBC686Q** messages.

For more information about attention processing in general, see **HELP ATTENTIONS.**

The DISCONNECT;GO Command String

As described elsewhere **(HELP COMMANDS DISCONNECT),** the **DISCONNECT** command can be used to disconnect your terminal from an cdf/XDC debugging session.
The **DISCONNECT** command, by itself, however, does not cause the user's program to resume execution. Instead, the program remains within the debugging session, and the session just waits for you or another user to connect back to it.

If you want **both** to disconnect from the debugging session **and** to let your program resume execution, then you will need to issue both a **DISCONNECT** command and a **GO** command simultaneously, i.e. **DISCONNECT;GO**

Caution: Reversing the two commands (GO;DISCONNECT) will not work. That's because when a program resumes execution (via the **GO** command), Z/XDC's input command stream is immediately flushed. So the **DISCONNECT** command will not be seen by Z/XDC, and your terminal will remain connected to the debugging session with the keyboard locked. (You can, however, use the **ATTN** key to get out of this. [See above.])

Help XDCSrver Cdf Tso

cdf/XDC exists primarily to allow users to conduct interactive Z/XDC debugging sessions with batch jobs and system tasks. However, a side benefit of that support is the ability to use cdf/XDC to connect to debugging sessions in TSO as well.

At first glance, it might seen that this would be a rather pointless capability. After all, TSO address spaces already have terminals connected to them, and Z/XDC already has comprehensive support for using TSO terminals in a variety of line mode and fullscreen ways.

Well it turns out that there is a circumstance under which being able to make Z/XDC use a terminal that is different from the TSO terminal is a very useful thing! That's when the program being debugging is itself a fullscreen terminal application.

When Z/XDC is being used to debug a program that itself issues fullscreen displays, that program's fullscreen display management and Z/XDC's fullscreen management will interfere with each other, with each program corrupting the other program's displays. When this happens, the best solution is to cause Z/XDC to use an entirely different terminal for its displays.

To cause Z/XDC to use a separate terminal for its displays, do the following:

- Before starting the debugging session, go to ISPF's Command Shell (=6), and issue the following command:
  TSO ALLOC FILE(xxxCDF) DUMMY REUSE
  where **xxx** matches Z/XDC's current command name (usually **XDC).**

- Now go to Z/XDC's Startup Panel and start your debugging session. Z/XDC will receive control, and your TSO terminal will lock up.

- Now go to **another** TSO terminal, and go to Z/XDC's Startup Panel.

- Type option **3** to go to cdf/XDC's Job Selection Menu. You should see your first TSO session listed there.

- Type **S** next to your first session's name to connect to the debugging session. You will now see Z/XDC's normal debugging session display.

- Proceed as you normally would: Issue your mapping commands, your breakpointing commands, etc.

- Eventually, you will probably issue a **GO** command. When you do, the following will happen:

  - Your debugging session's terminal will lock up.

  - Your program, running in the first TSO session, will commence to execute.

  - When your program issues its displays, they will appear on your first session's terminal, and it will unlock.

  - When your program ABENDs or reaches a breakpoint, Z/XDC will regain control, your first terminal will lock up again, Z/XDC's display will appear on your debugging session's terminal, and that terminal will unlock.

So your program and your debugging session will proceed in this fashion:

- When Z/XDC is in control, your program's terminal will be locked, and your debugging session's terminal will be unlocked.

- When your program is in control, your debugging session's terminal will be locked, and your program's terminal will be unlocked.

Help XDCSrver Cdf Logon

When a user program ABENDs or reaches a breakpoint, then if Z/XDC has been properly setup as the environment's recovery routine, then Z/XDC will receive control from

the System in order to process the ABEND (or breakpoint). When Z/XDC receives control in a batch job or system task, it will usually try to connect to cdf/XDC in order to find a programmer to talk to.

cdf/XDC is basically nothing more than a two-way communications line. When Z/XDC connects to cdf/XDC, it is connecting to one end of the line. It then has to wait for a programmer to connect to the other end. The length of time that Z/XDC waits is determined by your session profile. The factory default time limit is one hour. This time limit can be changed by the **SET SIGNONWAIT** command.

A programmer may logon to cdf/XDC either directly from a VTAM terminal or from a TSO session via the Z/XDC Startup Panel in ISPF.

- **From a VTAM terminal:**
  Use one or the other of the following VTAM logon commands:
  LOGON APPLID=XDC
  LOGON APPLID(XDC)

  Note, if you are using an XDC clone (i.e. a renamed XDC), then you will have to specify the clone's name in place of **XDC.** Example:
  LOGON APPLID=xxx
  LOGON APPLID(xxx)

  For more information about renaming Z/XDC, see **HELP XDCCLONES.**

  Note, if the **XDC** APPLID does not work, then check with your System Programmers. They may have changed the name of the APPLID from **XDC** (or from **xxx)** to something else entirely.

  In any case, once you have issued the **LOGON APPLID** command, you will then see cdf/XDC's Logon Panel. You must then type your TSO userid and password (or pass phrase) in order to complete your connection to cdf/XDC.

- **From a TSO session:**
  Start up ISPF and go to Z/XDC's Startup Panel in ISPF. From there, select option **3.** (This is the **cdf/XDC** option.)

  As you look at the Z/XDC Startup Panel, you will see a lot of input fields. Most of these you can ignore. (They apply to options 1 and 2, not to option 3.) The only input field that matters for option 3 is the last one. It reads: **Connect to cdf/XDC using your current TSO Userid? (YES/NO) ===>**

  If you fill in **YES,** then cdf/XDC automatically accepts your TSO logon identity, and so you are taken directly to the cdf/XDC **Job Selection Menu.**

  On the other hand, if you fill in **NO,** then you presumedly want to connect to cdf/XDC using a different identity, so you will then see cdf/XDC's Logon Screen.

When Cross Sysplex support has been activated in cdf/XDC, the debugging session can be running on any processor in a Coupling Facility connected Sysplex. Your TSO logon does not have to be to the same processor.

(Note, the location of Z/XDC's Startup Panel within ISPF is completely up to decisions made by the Systems Programmer who installed Z/XDC. We suggest **=D,** but check with him if you don't know how to find the Z/XDC Startup Panel.)

Once you have connected to cdf/XDC, you will then be presented with cdf/XDC's **Job Selection Menu.** This shows a list of background job debugging sessions to which you

are authorized to connect. Merely select the desired job. Then after an authorization recheck, cdf/XDC will hand you off to the job's debugging session.

If the debugging session is running on the same processor which your TSO session is logged on to, then the recheck will be under the covers. You will not see it. However, if it is running on a different processor, then you will see a cdf/XDC logon screen. This is because the system to which you are connecting requires its own verification of your identity. (This 2nd logon will occur regardless of whether or not you requested **USERID Propagation** way back on the Z/XDC Startup Panel in ISPF.)

From this point on you are connected to the background job in a fullscreen display mode that is very similar to a normal debugging session in TSO. You have available to you the full range of Z/XDC commands for debugging the background program.

Help XDCSrver Cdf COMMAnds

This topic describes user commands that either are unique to or are affected by cdf/XDC.

cdf/XDC also accepts commands from the System Operators. Those commands are not documented here, but the following Operator command can be used to display a description of them:

MODIFY xxxSRVER,HELP

The following user commands either are unique to cdf/XDC or require special consideration.

DISCONNECT - This command is unique to cdf/XDC. It causes the program being debugged to break the connection between it and the user. The program's debugging session returns to waiting for another user to signon to it, while the user is returned to cdf/XDC where he is again presented with a list of debugging sessions he can signon to. For more information, see **HELP COMMANDS DISCONNECT.**

END - The Z/XDC "END" command, depending upon operands, does not necessarily break the user's connection to the debugging session. It just causes the current ABEND in the current task of the background program to be percolated up to the next older ESTAE/I, if any (which could be Z/XDC again). If all older ESTAE/I routines fail to recover the ABEND, then the background program may be terminated by the ABEND, in which case the user's connection to the debugging session will then be broken, and control of the terminal will be returned to cdf/XDC or to ISPF.

On the other hand, if the **END** command does not lead to address space termination, the terminal's connection to cdf/XDC will persist. The keyboard will be unlocked or locked depending upon whether or not Z/XDC should happen to regain control in the address space. If the keyboard remains locked, then you may use the "attention" key to regain control and look around or to disconnect from the debugging session. For more information, see the following topics:

HELP COMMANDS END

HELP XDCSRVER CDF CONNECTION

GO - The **GO** command causes Z/XDC, in the background address space, to release control of execution and return it back to the user program. Since it is Z/XDC that talks to your terminal, not the user program, this action causes the terminal keyboard to lock up. Whether or not this lockup is noticeable depends upon how long it takes for the user program to reach another breakpoint.

If the user program simply continues execution without reaching a breakpoint, then the terminal keyboard will remain locked. You may use the **ATTN** key (or **PA1,** as appropriate) to break through the lockout. See **HELP XDCSRVER CDF CONNECTION** for more information.

GOT
GOX - When using cdf/XDC, the **GOT** and **GOX** commands behave identically to **GO.** Distinctions between **GO, GOT** and **GOX** arise only when debugging programs running within TSO and when communication to said debugging session is not via cdf/XDC.

TSO - If the user is connected to cdf/XDC from a TSO session (i.e. if he has not signed directly on to cdf/XDC via VTAM), then this command can be used to run an arbitrary TSO command within the user's TSO address space. The command is NOT transmitted to the background debugging session. The usage of this command is very similar to Z/XDC's **TSO** command during a normal debugging session. See **HELP COMMANDS TSO** for more information.

SPLIT
SWAP - If the user is connected to cdf/XDC from a TSO session (i.e. if he has not signed directly on to cdf/XDC via VTAM), and if he has made the connection from within ISPF, and if he has **SET ISPF** in effect, then these commands will cause the ISPF running in the user's TSO address space to swap to other work within ISPF. In effect, the cdf/XDC debugging session with the background job will be suspended until the user tells ISPF to swap back to the debugging session.

Help XDCSrver Cdf Fasm

cdf/XDC and Z/XDC's "Foreign Address Space Mode" ("FASM") are two mechanisms that Z/XDC provides for debugging programs in other address spaces. cdf/XDC and FASM, on the surface, may seem to be similar, but in reality, they differ from each other in several very important ways:

Startup:

To use FASM, you must start an authorized debugging session at your TSO terminal and then issue a SET ASID command to target the address space of your choice. To start an authorized debugging session, use Z/XDC's Startup Panel in ISPF, or go to ISPF's Option 6 and type "XDCCALLA IEFBR14". (Note, system security might not permit you to start an authorized debugging session.)

To use cdf/XDC, you must create an interface to Z/XDC in your batch program, and then cause your program to ABEND or reach a breakpoint in order to pass control to Z/XDC in the background address space. Then you must connect to that debugging session from your terminal.

Tip: One way to create an cdf/XDC debugging session is to start a FASM session, then use Z/XDC's HOOK command to create an Z/XDC interface in the targeted program.

Where Z/XDC is running:

With FASM, Z/XDC runs only in your local TSO address space. All commands that you issue are executed locally in the TSO address space. All displays that you request are accomplished by Z/XDC via various cross memory access techniques.

With cdf/XDC, Z/XDC runs in "Local Address Space Mode" within the background address space. Most commands that you issue are executed in the background address space. All displays are transmitted from there to your terminal via various cdf/XDC and VTAM service routines.

Eligible Target Address Spaces:

With FASM, **any** address space (security permitting) can be "debugged". No setup is required in the targeted address space. Also, no cooperation is required from the targeted address space. In fact, the targeted address space can even be "dead": looping, waiting, rampaging, whatever.

With cdf/XDC, an interface to Z/XDC must exist in the targeted address space, and the user's program must have ABENDed or reached a breakpoint and, therefore, passed control to Z/XDC. In other words, the targeted address space must cooperate fully with the debugging session.

Note, when in Foreign Address Space Mode, the HOOK command can be used to create an cdf/XDC connectable debugging session in the targeted address space.

Invasiveness:

Since FASM does not require any cooperation from the targeted address space, FASM can be used to produce displays without affecting the address space's execution progress in any significant way. (FASM also supports HOOK'ing and ZAP'ing which, of course, can be quite invasive.)

Since cdf/XDC requires Z/XDC to be in control in the targeted address space, cdf/XDC is very invasive. User program execution will be stopped whenever Z/XDC is in control (due to an ABEND or a breakpoint).

Security and access levels:

FASM security allows three levels of access to an address space: "read/write", "read only", and "none". Access is controlled by system security rules that define which users have which level of access to which jobs. "Read/write" access permits displays, hooks, and zaps. "Read only" access permits only displays, no hooks or zaps.

cdf/XDC security allows only two levels of access to an address space: "read/write" and "none". Since cdf/XDC usage already requires that the Target Address Space be modified at least enough to insert an Z/XDC interface, providing a "read only" level of access does not make a lot of sense.

Limitations:

With FASM, you can set hooks into the targeted address space, but you **cannot** set breakpoints there. In other words, you cannot conduct a debugging session in a foreign address space via FASM mode.

What you **can** do is set a hook. Then when the program that is running in that address space reaches the hook, a new debugging session will be started in that space, and you can then use cdf/XDC to connect to it.

So in FASM mode, you **can** issue the following kinds of commands:
- Display generating commands: LIST, FORMAT, DISPLAY, and SHOW.
- Display support commands: MAP, DMAP, USING, EQUATE, etc.
- Terminal support and profile support commands: SET KEYS, SET COLORS, scrolling commands, PROFILE, etc.
- Certain zapping commands (ZAP, and POST) when read/write access is permitted.
- The HOOK command for creating a full blown debugging session.

What you **cannot** issue are:
- Breakpointing commands: AT, TRAP, TRACE, etc.
- Other altering commands: LOAD/DELETE modules, GETMAIN/FREEMAIN storage, etc.

With cdf/XDC, there are no restrictions on the debugging session. All of Z/XDC’s commands and capabilities are fully supported.

Authorization:

FASM requires you to start an authorized debugging session in your TSO address space.

cdf/XDC does not require authorization either in your TSO address space or in the targeted background job.

Uses:

Use FASM for investigating other address spaces. It is an excellent tool for:
- Examining an address space "on the fly".
- Examining a running address space to learn such things as its subtask structure, the programs that it is running, the control blocks that it uses and how they hang together, etc.
- Sampling the execution path of any given program running under any task.
- Examining a failed (but still existing) address space for the point of failure and even sometimes repairing the failure allowing the address space to resume normal processing.

In other words, if you have questions about what might be going on within any address space, then FASM is a very powerful tool for answering such questions.

Use cdf/XDC for program development of batch jobs, system tasks, and TSO/ISPF

fullscreen display programs. cdf/XDC provides full support of all of Z/XDC's many features.

Use cdf/XDC and FASM together. You not only can use the HOOK command from a FASM session to create an cdf/XDC session, but you also can use the HOOK command to reactivate an cdf/XDC session that already exists but that you have lost control of. Example: Perhaps you have set a breakpoint, and issued a GO command, but contrary to expectations, your program took a different path and so never reached the breakpoint, thus leaving you locked out of the session. You can use Z/XDC's FASM mode not only to find out what's going on, but also, if desired, to reestablish the cdf/XDC session. Simply do the following:

- Switch to another terminal (or just attention out of the locked cdf/XDC session).
- Start an authorized debugging session in TSO (example: XDCCALLA IEFBR14).
- Switch to FASM mode (SET ASID jobname).
- Look at the subtask structure (LIST TASKS).
- Look at the Request Blocks queue (LIST RBS TCB#n).
- Look at your program's registers (LIST REGS RB#n).
- Look at storage (FORMAT, SHOW, DISPLAY, etc.).
- Decide what to do next.
- Maybe use the HOOK command to throw control back to Z/XDC within the address space that you are debugging.

Help XDCSrver Cdf Recovery

Sometimes during a debugging session, you might be interrupted by a communication problem, a network outage, a line drop, whatever. Normally, when this occurs, VTAM will be notified, and it, in turn, will notify cdf/XDC. Then cdf/XDC will reset the connection, reissue its "AWAITING USER SIGNON" WTOs and WTOR (DBC640Q), and then wait for the user to connect back in to resume the debugging session.

However, sometimes a line drop can occur in which VTAM is **not** notified. When this occurs then cdf/XDC also is not notified, and so it does not reset the line, and the user is left with no immediate way to reconnect to and resume his debugging session.

Well, what needs to be done is to manually tell VTAM to reset the line. and that will cause cdf/XDC to reset its view of the connection, reissue its "AWAITING USER SIGNON" WTOs and WTOR (DBC640Q), and then wait for the user to connect back in.

But the only way to do this is to issue a series of System Operator Commands, so you will need either to have access to a console interface or asked for cooperation from your Data Center's Operations staff.

Anyway, here's the commands that need to be issued.

- **F xxxSRVER,CDF DISPLAY ALL**
  ("xxx" represents Z/XDC's current clone name. Usually, "xxx" is "XDC", but it could be just about anything. For more information, see HELP XDCCLONES.)

  The response to this command is a series of DBC647I messages listing all of the VTAM APPLIDs that are connected to the xxxSRVER address space. Their names

should all start with xxx. In fact, the first one should be exactly xxx (the clone name).

Choose one of the APPLIDs (such as the first one), and then issue the following command:

- **D NET,ID=xxx,E**
  This is a VTAM command that produces a display of 20+ ISTnnnI messages describing the APPLID named xxx.

  Look for the message: **IST231I APPL MAJOR NODE = name.** This gives the name of the major node that owns the xxx APPLID. Usually, that name will be xxxSRVER, but it might be something else. In any case, use that name in the following command.

- **D NET,ID=name,E**
  This will produce a series of IST080I messages listing the status of all of the APPLIDs owned by this node. Several of those APPLIDs will be named **xxxTFSnn** (where nn is a sequence number).

  The status of most of the APPLIDs will be CONCT. That's IBM-speak for pending a user connection but not actually in use yet. Ignore those.

  Hopefully, at least one of the xxxTFSnn sessions will have a status of **ACT/S.** One of those will be the one that was being used by your debugging session.

  Pick one of the ACT/S sessions to reset. If there's more than one to pick, then do the following to sort out which one was yours.

  - **D NET,ID=xxxTFSnn,E**
    Look towards the bottom of this display for IST635I messages showing other APPLIDs that are involved with the same VTAM session. You should see two APPLIDs: xxxSIDE and another. If you do **not** see a xxxTSOnn APPLID, then this xxxTFSnn may be the APPLID you want to reset.

    On the other hand, if you do see a second APPLID listed, then issue **D NET,ID=name,E** to display its information. Examine the resulting display to see if it is/was your old session or someone else's.

  Once you've chosen an **xxxTFSnn** session to reset, do so with the following command.

- **V NET,ID=xxxTFSnn,INACT,I**
  This will deactivate the VTAM connection, causing cdf/XDC reset its view of the connection, reissue its "AWAITING USER SIGNON" WTOs and WTOR (DBC640Q), and then wait for the user to connect back in to resume the debugging session.

- **V NET,ID=xxxTFSnn,ACT**
  This will restore the xxxTFSnn APPLID to the "connectable" state (CONCT).

Bouncing cdf/XDC

If the above process does not work, then your only choices left will be either to cancel the job you're trying to debug, or bounce the cdf/XDC subserver as well as it's entire VTAM node (usually named xxxCDF). The following operator commands will accomplish this:

```
F xxxSRVER,MANAGER HALT CDF
V NET,ID=xxxCDF,INACT,I
V NET,ID=xxxCDF,ACT
F xxxSRVER,MANAGER START CDF
```

After each command, wait a moment or two before issuing the next.

Help XDCSrver Cdf CRosssysplex

When appropriately installed, cdf/XDC supports users connecting to debugging sessions located on any processor in a Parallel Sysplex (i.e. a Coupling Facility connected Sysplex). This is called Cross Sysplex Support. The following are brief remarks about activating Cross Sysplex Support in cdf/XDC.

Installation Requirements

The following requirements must be met:

- **Coupling Facility:** The Sysplex must be a Parallel Sysplex with the processors connected together via a Coupling Facility. A Basic Sysplex is not supported.
- **z/OS Level:** The z/OSs participating in the Cross Sysplex Support must be at the R1.13 or newer levels. R1.12 and older levels are not supported.
- **Z/XDC Level:** z/XDC must be at release z2.1 or newer. The z2.1 release must be at maintenance level Z21-1505A or newer.
- **Sharing:** The following services and facilities must be shared to participating Systems:
  - **Global Resource Sharing** (GRS for ENQs)
  - **Global Resources:** (a VTAM facility)
  - **ISTGENERIC:** This VTAM structure must be properly defined to the Coupling Facility.
  - **&SYSCLONE:** This System Symbol must be unique for each Sysplex participant.
- **Clone Names:** The participating Z/XDCs must all have the same clone name. (As usual, differing clones may participate independently in their own instances of cdf/XDC.)

Installation Elements

There are three elements involved in setting up cdf/XDC: A JCL PROC, a //SYSIN

parameter file, and a VTAM Major Node definition. Coordinated, Factory Default samples of all three elements are distributed with Z/XDC via the **hlq.XDCV31.XDCSRVER** library. Many shops can install and run these samples without change. (But to activate Cross Sysplex Support, you are going to have to make some changes.) The samples are:

- **XSRVPROC:** This a JCL PROC that contains sample JCL for running server/XDC (the XDCSRVER job). cdf/XDC runs as a subserver within server/XDC. The JCL is pretty simple. It can be run with little or no change.

- **XSRVPARM:** This is a parameter file for all of the subservers that run within server/XDC. The section for cdf/XDC parameters is headed by a **<CDF>** statement.

  XSRVPARM contains extensive commentary documenting all of the possible parameter statements and, in particular, documenting what you have to do to activate Cross Sysplex Support.

- **XSRVVTAM:** This contains a sample VTAM Major Node definition. It is just a collection of VTAM **APPLID** statements that define the necessary elements of the Major Node that is needed by cdf/XDC. XSRVVTAM needs to be copied into VTAM's VTAMLST library. The suggested member name is **XDCCDF.** Then the node can be activated via an operator command: **V NET,ID=XDCCDF,ACT**

  Most customers can use XSRVVTAM without making any changes to it. But when you wish to activate Cross Sysplex Support, you'll need to make changes. Each participating System must use its own, uniquely defined Major Node.

  Detailed rules are presented in commentary found in the **XSRVPARM** member.

Help SUpport

ColeSoft wants to make sure that you get good use from the Z/XDC product. If you are unable to answer your questions by consulting the available documentation and this Built-in Help subsystem (even if it's because the doc is just too long to read), please feel free to send us an email. Our contact information can be found at **HELP SUPPORT CONTACTUS.**

Reporting Problems

If you want to report a problem, it would be helpful if you would send the following supporting documentation:

- Please send a **session log** showing the problem that you are reporting. (Session logs usually are found on JES2/JES3 spool in SYSOUT class X.)

- Please include, either in the session log or in the session log of a separate session, the output of the **SYSINFO script.**

- If Z/XDC has ABENDed, then please include all **SYSLOG messages,** especially the summary dump messages (DBC913T, DBC914T, DBC916T, etc.).

- Also, save (but do not send yet) any **ABEND dumps** that might by relevant. (We might need them, we might not.)

- For more information about sending us dumps and such, see **HELP SUPPORT SENDINGUSDUMPS.**

Maintenance

ColeSoft uses maintenance updates both to publish maintenance (fixes, corrections and the like) and to publish new functionality. That subject is discussed in depth at **HELP MAINTENANCE.** Both the maintenance process and content can be found there.

Subtopics Index

The following additional information is available. Type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

FEATURESANDCAPS	- A discussion of Z/XDC's various Licensed Features
HLQ	- Your local names for our hlq.XDCV31.XDCwhatever Product Libraries
MANUALS	- A description of the downloadable PDFs available for Z/XDC
ONLINE	- About our website, our YouTube pages, and about getting doc, training videos, expert help and the like
CONTACTUS	- How to contact us
SENDINGUSDUMPS	- About sending dumps to us
SUPPORTSTATEMENT	- The hardware and operating systems within which Z/XDC release v3.1 can be used
LEGALNOTICE	- Copyright notice and warranty statement
CREDITS	- Giving credit where credit is due

Help SUpport Featuresandcaps

The **Z/XDC** Product consists of a base product (the **Z/XDC Engine)** and several individual Licensed Features:

- **base/XDC**
  This is Z/XDC's core product. It is also called the **Z/XDC Engine.** It is always present. It provides all of Z/XDC's functionality that is not reserved for the various Licensed Features. In particular, with just base/XDC installed, you can do the following:
  - Load maps of load modules (linkedit maps) to learn the structure of a load module
  - Disassemble machine code
  - Issue nearly all of Z/XDC's various **LIST** commands for displaying registers, system states, system structures and Z/XDC's options, settings and lists
  - Issue nearly all of Z/XDC's various **SET** commands for changing options, settings, states and such

- Issue the **ZAP** command for altering registers and storage
- Issue the **EQUATE** command for creating symbols for labeling locations in storage
- Load, change and save session profiles
- Debug problem state programs
- Debug multitasking programs
- Debug problem state/key PC routines
- Debug system and product exit routines that run non-authorized

And so on...

- **asm/XDC Feature**
  This Feature licenses Z/XDC's additional support for debugging Assembler programs. When licensed, asm/XDC enables the following:
  - The ability to load SYM data based maps and dsects for Assembler programs
  - The ability to load ADATA based maps and dsects for Assembler programs

- **c/XDC Feature**
  This feature licenses Z/XDC's support for debugging programs written in the XL C/C++ and Metal C Language. When this feature is licensed, the following Z/XDC capabilities are activated:
  - The ability to load **Source Image Maps** for programs written in XL C/C++ and Metal C.
  - The **STEP** command for stepping execution into, out of, over and through an XL C/C++ and Metal C subroutine.
  - The **LIST VARIABLES** command for displaying High Level Language variables, structures and arrays.
  - Various other commands relating to the display and management of High Level language variables.

- **auth/XDC Feature**
  This Feature licenses Z/XDC's support for debugging programs that are running with APF authorization and/or in supervisor state and/or with a system execution key. When licensed, auth/XDC enables the debugging of the following:
  - Programs running APF authorized (i.e. with JFCBAUTH=1 set)
  - Programs running in supervisor state
  - Programs running with a system execution key (keys 0 through 7)
  - System exit routines (that run authorized)
  - SVC routines
  - PC routines (that run authorized)
  - SRB routines
  - FRR protected programs
  - LOCK'd routines
  - Programs running in other address spaces (Foreign Address Space Mode)

The Licensed Features are completely independent of each other. Technologically, they can be licensed in any combination that makes sense for your programming needs. (ColeSoft's pricing policies may limit the permitted choices.)

Legacy Licensing

Feature Licensing was introduced in 2009 with release z1.10 of Z/XDC. In the releases prior to z1.10, the capabilities of the Z/XDC product were equivalent to the following set of **Legacy Features:**

base/XDC
asm/XDC
auth/XDC

Accordingly, as of z1.10's release date, existing customers were automatically licensed to use the Legacy Features Set. Further, the pricing of the Legacy Features Set was set so that existing Customers would see no price change.

For more information, see **Sales** in HELP SUPPORT CONTACTUS.

Concurrent Access Permits (CAPs)

Licensed features are controlled by **CAPs.** There are as many types of CAPs as there are features. One CAP allows one programmer to use the corresponding feature. When Z/XDC is licensed for use, the license includes an agreed upon number of CAPs for each Feature that is licensed.

Each copy of Z/XDC is customized by defining the numbers and types of CAPs that the customer has licensed. This information is stored into Z/XDC via the **LICENSE** command. See HELP COMMANDS LICENSE for more information.

CAPs limit the number of programmers across your systems that are allowed to use a given Feature simultaneously. CAPs are assigned to a debugging session upon first use of its associated Feature. Once assigned, CAPs normally remain assigned for as long as the program being debugged continues to run. But if you want to free up a session's CAPs before the program ends, you can use the **GO RELEASECAPS** command to do that. See HELP COMMANDS GO for more information.

CAPs are assigned to a debugging session upon the first occurrence of the following events:

- **base/XDC:** At the start of a debugging session.
- **auth/XDC:** At the point at which a debugging session finds itself to be running authorized. (This usually occurs at the start of a debugging session.)

- **asm/XDC:** Upon first use of the **MAP** or **DMAP** command to load an ADATA or SYM data map of a csect or dsect.

- **c/XDC:** Upon first use of the **MAP** command to load a source image map of a C, C++, Metal C or Systems/C program.

Help SUpport Hlq

Product Library Names

Throughout the Help, Product Library names are referred to as **hlq.XDCV31.XDCwhatever,** where:
- **hlq** is the library name's High Level Qualifier.
- **whatever** is a library's unique name.

Every customer will create the Z/XDC Product Libraries using their own, in-house High Level Qualifier. You will have to ask your Systems Programmer what the hlq is set to at your shop.

Help SUpport Manuals

All of the manuals documenting Z/XDC can be found at our web site: **colesoft.com/pdfs.** They are all in PDF format, and they all can be downloaded as needed by Z/XDC customers.

Broadly speaking, the manuals fall into two categories:
- Those that are built upon the content of this Built-in Help
- Those that are not

Manuals Based on the Built-in Help

The following manuals are direct extractions of various branches of this Built-in Help

- **Maintenance and New Functionality Guide:** This document is intended to let customers stay up to date whenever a new update is installed against Z/XDC. The Guide thoroughly documents what is new and changed.

- **Users Guide:** This manual contains numerous tutorials about various aspects of using Z/XDC to debug various kinds of programs running in various z/OS execution environments.

- **Commands Reference:** This manual is a comprehensive reference documenting all Z/XDC commands. It provides:
  - Syntax details
  - Usage information
  - Usage examples

- **Messages Reference:** This manual gives exhaustive documentation for all numbered **DBCnnnc** messages. The discussions tend to be exhaustive.

The manuals published at colesoft.com/pdfs are updated to maintenance level **DBC-2506A** or newer.

Additional Manuals

The following manuals provide additional information about Z/XDC

- **Z/XDC Primer:** This is a task-oriented, how-to guide for using Z/XDC.
- **Z/XDC Quick Reference:** This gives brief descriptions and syntax diagrams for all Z/XDC commands.
- **Z/XDC Installation Guide:** This provides detailed information about installing Z/XDC onto your system. It is intended for SysProgs who are responsible for installing and maintaining mainframe software.

As noted above, all of these manuals are in PDF format can be downloaded from **colesoft.com/pdfs.**

Help SUpport Online

ColeSoft supports an increasing number of Online Services for our Customers. We are on Facebook and YouTube. And of course we have our own home at **colesoft.com.**

You can use our several services to get doc, get fixes, and get temporary licenses. You can get help and give help, and you can watch training videos. For details, check out the following topics. You can type an **H** at the left to select directly, or use **HELP *NEXT (F11)** to proceed sequentially. Use **HELP *FORWARD (F5)** to skip.

WEBSITE - About our home page
FACEBOOK - About our Facebook presence
YOUTUBE - Where you can find Z/XDC training videos

Help SUpport Online Website

Our home website is **colesoft.com.** The area of the website that is devoted to Z/XDC can be reached directly via **xdc.com.**

colesoft.com is intended to be an introduction to us and our products for potential customers searching for the kinds of things we do. As such, its content is pretty static.

However, **colesoft.com** does provide several services that are useful to existing customers:

Maintenance Downloads

All of our product maintenance is published on our website's Maintenance Page. When you receive notice of new maintenance, this is where you go to download it. For details, see **HELP MAINTENANCE SUPPORT.**

Temporary Licenses

Sometimes it arises that you may need a temporary activation code for Z/XDC. Perhaps you need to conduct a DR test. Perhaps you need to recover from an actual disaster. Whatever.

colesoft.com has a self-service page where you can create a 14-day activation code for Z/XDC:

- Just go to **colesoft.com.**
- Select the **Support** drop down.
- Select **Activation Codes.**
- Then fill out the form and follow the instructions.

You will immediately receive a 14-day activation code. It's that easy!

Z/XDC Manuals

Currently, there are seven manuals documenting various aspects of Z/XDC in various ways. They all can be downloaded in PDF form from our website. For more information, see **HELP SUPPORT MANUALS.**

Module Mapping Data Extraction Utility

Many of our customers are ISVs (Independent Software Vendors) who from time to time need to troubleshoot dumps of their products received from their customers. If our customers use **dump/XDC** to help examine the dump, they can ask their customers (the downstream customers) to send them mapping data for the load modules contained within the dump.

The mapping data is extracted and compressed by a utility name **XDCMMEF.** That utility and its usage doc are publicly available and can be freely downloaded (with no obligation) from our website at **colesoft.com/public/downloads/MMEF.** See **HELP UTILITIES XDCMMEF** for more information about **XDCMMEF.**

z/OS Control Block Maps for Older Releases of z/OS

When using dump/XDC to examine a dump, if the dump was taken from a system running an older release of z/OS, then the examiner is going to need system control block maps that match the z/OS release level from the dump.

colesoft.com/support/downloads/ZOSMAPS contains downloads of system control block

mapping data for all releases of z/OS going back to z/OS **R1.6!** For more information, see **HELP DUMPS MAPPINGDATA ZOSCBLOCKS.**

Help SUpport Online Facebook

ColeSoft has a Facebook page at **facebook.com/colesoftware.** This is where we publish "softer" information... Announcements of our company activities and such.

We also make technical announcements here when needed, but our primary place for the hard stuff is our blog at **colesoft.com.**

Help SUpport Online Youtube

ColeSoft has created a number of Z/XDC training videos and posted them on YouTube. Our channel is named **youtube.com/colesoftware.**

Currently, there are eleven videos. They introduce Z/XDC and provide detailed discussions and examples for using various aspects of Z/XDC in various ways. Please check them out.

Help SUpport COntactus

If you require Technical support, by far the fastest way is to email us at support@colesoft.com.

That eaddress automatically fans out to several of us here at ColeSoft.

We also can be reached for various purposes in any of the following ways:

Customer Services: office@colesoft.com
WEBSITE: colesoft.com
FTP SITE: ftp.colesoft.com

Sales: 540-456-8210
EMAIL: sales@colesoft.com
WEBSITE: colesoft.com

Snail Mail: ColeSoft
440 Monticello Ave
Ste 1802 PMB 380764

Norfolk, Virginia 23510-2670
USA

Thank You,
Dave Cole

Help SUpport SEndingusdumps

It is ironic that even though Z/XDC is as powerful as it is, we nonetheless sometimes (but not always) have to rely upon dumps to debug customer problems that we cannot recreate here.

But when you've got a dump, please do not just send it to us automatically. Often we can solve the problem with the information listed in HELP SUPPORT. If it turns out that we do need the dump, we will ask for it.

Support Documentation

Before sending us a dump it would be helpful if you would first send the following supporting documentation:

- Please send a **session log** showing the problem that you are reporting. (Session logs usually are found on JES2/JES3 spool in SYSOUT class X.)
- Please include, either in the session log or in the session log of a separate session, the output of the **SYSINFO script.**
- If Z/XDC has ABENDed, then please include all **SYSLOG messages,** especially the summary dump messages (DBC913T, DBC914T, DBC916T, etc.).
- Also, save (but do not send yet) any **ABEND dumps** that might by relevant. (I may need them, I might not.)

Getting a Dump

So if we ask for a dump, then please send a compressed copy to our FTP site. Below, I will explain how to do that. But first, ...

When an unexpected ABEND occurs, Z/XDC does the following:

- It produces a "summary dump" (headed by message DBC913T) describing the ABEND.
- It then either issues an SDUMPX macro to produce an SVC dump, or it sets the SDWAREQ flag to ask the System's Recovery/Termination Manager (RTM) to produce a dump. (Z/XDC will not do both.)
- It then percolates the ABEND, thus ending the debugging session.

When percolating the ABEND back to RTM, if an XDC SDUMP has not been requested, (i.e. if an //xxxSDUMP allocation is not present, see below), then Z/XDC will set the SDWAREQ flag on to ask RTM to produce a dump. Whether or not a dump is actually produced, however, depends both upon your local Data Center dump management controls and upon what subsequent recovery routines might do with the SDWAREQ flag before RTM

has a chance to see it.

But if you do not want to rely upon SDWAREQ=1, you can instead cause Z/XDC to explicitly take its own dump. To do so, then **prior to** an unexpected ABEND (i.e. all the time) add an xxxSDUMP allocation to your debugging session (where "xxx" matches Z/XDC's current clone name, usually "XDC"):

- If debugging a batch job, add a **//xxxSDUMP DD DUMMY** DD-card to your batch job's JCL.
- If debugging a program running in TSO, issue a **TSO ALLOC DDNAME(xxxSDUMP) DUMMY REUSE** command.

Substitute Z/XDC's current clone name (usually "XDC") for "xxx".

When an xxxSDUMP allocation is present, then the recovery routine, one way or another, will issue an SDUMPX macro to create a full dump of both common storage and the current home address space. (But whether or not a dump actually gets produced will depend upon your Data Center's local dump management controls.)

(Note, Z/XDC expects the //xxxSDUMP to be a dummy allocation. Z/XDC will never open that file, and it will never write a dump to it. Instead, the SDUMP service will always be directed to write the dump to system dump datasets.)

When Z/XDC issues the SDUMPX macro, it will also turn the SDWAREQ flag **off** to keep the RTM from producing a redundant dump.

Compressing the Dump

AMATERSE is a good, IBM supplied program for compressing a dataset or a library into an Internet transmittable container file. To run AMATERSE, use the following sample JCL as a guide:

```
//*
//RESET   EXEC PGM=IEFBR14
//OUTFILE  DD  DSN=compressed.xdcdump.dataset,
//             UNIT=SYSALLDA,SPACE=(TRK,0),DISP=(MOD,DELETE)
//*
//COMPRESS EXEC PGM=AMATERSE,PARM=PACK,REGION=0M
//SYSPRINT DD  SYSOUT=*
//SYSUT1   DD  DISP=SHR,DSN=original.xdcdump.dataset
//SYSUT2   DD  DSN=*.RESET.OUTFILE,UNIT=SYSALLDA,
//             DCB=(RECFM=FB,LRECL=1024,BLKSIZE=0),
//             SPACE=(TRK,(5000,1000),RLSE),DISP=(,CATLG)
```

FTP'ing the Dump to Us

ColeSoft has an FTP site named **ftp.colesoft.com.** Note the following:

- If your mainframe is connected to the Internet, then you can send the compressed dump directly from your mainframe to our site. Otherwise, you will have to download it to your PC and send it from there instead.
- If your mainframe does not have a Domain Name Server (DNS), then you can use our IP address instead: 216.30.182.253.

In all cases, for sending your file, here's an overview of what you need to do:

- Decide what you want your file to be named when it's uploaded to our ftp site.

(Important! Your **filename *must* have "XDC"** somewhere within it. See below for the complete rules.)
- Perform an "anonymous login" to our site.
- Navigate to the **incoming** directory. ("incoming" must be all lowercase.)
- Upload your file in **binary** format.
- Log out.
- Notify us via email.

Choosing a File Name for Your Dump

If the **ftp.colesoft.com/incoming** directory already has other files within it, you will not be able to see them, so when choosing a file name for your compressed dump, please follow these rules:

- Please do **not** use simple names such as "XDCDUMP". That doesn't really tell us much about what you're sending or who's sending it, and it could have a higher potential for conflicting with files sent from other customers.

- Please **do** include your company's name and/or your own name into the file name (mainly to better identify the file and to make it reasonably likely that the name will be unique).

- Please make sure that the name includes the string **xdc** somewhere (anywhere) within it. **This is required** because all files that do not have **xdc** within their names are automatically **purged** from the site. (We've had to do this because there are a large number of robots wandering around the Internet storing random files at "anonymous-enabled" ftp sites such as ours.)

Sample FTP Job

For your convenience, we have provided a sample FTP job for sending dumps (and whatever) to us. It is named **$FTPUP,** and it can be found in **hlq.XDCV31.XDCJCL.** Just follow the instructions provided in the job's commentary.

Remember: Once you've uploaded your file, don't forget to send us an email to tell us that it's there. You should send your email to whomever's been working with you on the problem.

Help SUpport SUpportstatement

Z/XDC release v3.1 will run on releases R2.2 through R3.2 of IBM's z/OS Operating System.

Z/XDC release v3.1 does, of course, recognize and support all z/Architecture machine instructions up to and including those documented in in the **-13** edition (for the Z16 machine) of IBM's **z/Arch Principles of Operation** (SA22-7832).
However, **internally,** Z/XDC uses only those machine instructions documented in the **-06** edition of that manual.

Z/XDC release v3.1 will run on any **Z10** or newer machine.

Z/XDC includes limited support for **CICS.** When loading modules into storage, CICS does not use z/OS services (LOAD and friends). Instead, it uses its own OCO mechanisms and control structures. Consequently, Z/XDC has to use special logic for knowing about load modules within CICS.

Currently (09/22/2022), Z/XDC supports identifying load modules within CICS releases 4.2 thru 5.6.

Help SUpport Legalnotice

Z/XDC, COPYRIGHT (C) IZZI SOFTWARE - 2010-2026. ALL RIGHTS RESERVED

Help assembly date - 09/03/26

Z/XDC and its documentation (collectively, "Product"), including copies thereof, are the confidential and proprietary property of ColeSoft Partners, Inc. ("Owner"). The Product may be used only by those organizations that are licensed by Owner for such use and only in the manner so licensed. The Product, its programs, and its documentation may not be published, reproduced, distributed, or made available to third parties for any purpose without the expressed written permission of Owner; however, a reasonable number of copies may be made of the documentation (including the copyright notices and proprietary legends thereon) as is necessary for the legitimate use of the Product within a licensed organization.

Except as may be otherwise expressed in a signed licensing agreement between Owner and Customer, Owner makes no representations or warranties, expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose, the warranty of freedom from rightful claims by way of infringement and the like, and any warranty as to accuracy.

WARNING! Z/XDC is a powerful tool for dynamically locating and correcting malfunctions in actively executing user programs and operating system programs and subroutines. Accordingly, it is inherent in its design, that unless the use of the Product is properly controlled, then under certain conditions a malicious or careless user can use the Product to alter, subvert, counterfeit, damage or otherwise disturb the normal execution of user programs or system routines including, under certain conditions, both its own and system security routines.

Therefore, even if advised of the possibility of damages, under no circumstances shall Owner be liable for any loss or damage whatsoever (including death) arising from the Product, whether such loss or damage be direct, indirect, consequential, special or otherwise. Further, Owner shall not be obligated to indemnify any user of the Product in any manner for any loss which the user or anyone else may experience, of any kind or nature, arising out of the use or misuse of the Product.

The Extended Debugging Control products are marketed by ColeSoft Marketing, Inc. with its principal office Charlottesville, Virginia. If you would like more information, please contact **sales@colesoft.com.**

Our physical address is:
ColeSoft
414 3rd Street NE
Charlottesville, Virginia 22902

For Technical Support, please see **HELP SUPPORT CONTACTUS.**

Help SUpport CRedits

A Brief History

I (Dave Cole) started developing what is now called Z/XDC back around 1978 or so. Back then, it was called **DBC** (my initials), and it worked in that antique operating system now generally called MVS/SP.

Then in the early 1980s, IBM introduced a new version of the Operating System named MVS/XA which featured the introduction of 31-bit addressing, something which DBC was completely unprepared to handle. So I had to totally revise the product, and the result was the first release of **XDC** in 1984.

Later, when z/OS and 64-bit addressing arrived on the scene (in 2002), Z/XDC adapted, and continued moving into the future.

Starting in 1978 or so, for the first 10 years of development, I worked solo. But in 1988, I quit my day job, I teamed up with Bob Shimizu, and I've committed myself ever since to the full time development of what is now known as **Z/XDC.**

Major Contributors

In the years since then, a number of people have come and gone. For a complete list, see commentary in the **#DBCVRSN** macro in the hlq.XDCV31.XDCMACS library.

Sadly, I cannot remember all of the contributors, but there are some that I'll never forget. Here is a partial list of those who have made major contributions to the code.

Michael Lewis

Mike came to me in the early 1980s with the idea that XDC desperately needed to work as a fullscreen display program. (Before Mike, XDC and DBC were strictly linemode devices.) Foolishly, I was skeptical. A year later he came back to me with an initial implementation. This time I was impressed.

Next to my own, Mike's code remains the single largest contribution to the product. His work included fullscreen display support, the Cross Domain Facility, session profile management, and session logging.

Mike left ColeSoft around 1993 or so, but returned in 2010 to engage in major development projects. His current job is to develop and maintain our **c/XDC** Licensed Feature. This provides source language support for LE C and C++, Metal C from IBM and Systems/C from Dignus.

Mike can be reached at **mike@colesoft.com.**

Jeffrey Smith

Jeff worked for ColeSoft from 1997 to 2000. In that time, he made numerous changes throughout the product. His major contributions included cross memory support and license control data management.

On the business side, Jeff converted our product licensing model away from perpetual licensing with an annual maintenance subscription to what it is today, an annually renewed lease priced according to usage levels instead of machine capacity. Given that a few large software houses were busily gobbling up all our customers, without Jeff's vision in this area, we probably would not have survived.

Frank Chu

Frank came to ColeSoft in 1998 as our local PC and network jockey, a job at which he excelled. After a couple of years, Frank began to take over the mainframe Systems Programming chores that, up to that point, had been mine. Again, he exceeded all expectations.

In about 2004, Frank added a new hat to his collection: "XDC Level Two Technical Support Technician". Questions that Bob can't answer get bumped to Frank. And when a more than average complex issue arises, Frank works closely with me to resolve all problems as quickly as possible.

To this day, Frank has continued to excel at Customer Technical Support. He is a master of dump analysis and of creating in-house test cases for even the most complex issues that come in our door.

More recently, we have begun development projects aimed at taking Z/XDC out of the TN3270 prison and into the wonderful world of GUI enabled interfaces. Frank is a central player in that effort.

Frank can be reached at **frank@colesoft.com.**

Bob Shimizu

Bob joined up with me in 1988 and remained until 2024. He didn't write a single line of code, but his commitment and dedication to ColeSoft was every bit as strong as my own. His contributions were, in many ways, even more important than mine: Making sales and creating lasting customer relationships!

Bob did something that is rare among salesmen. Not only did he learn the product to a technical depth that few of his profession are willing to plumb, but he also took on additional roles that were key to our survival and success.

Peter Morrison

Peter came onboard here in 2017. He is an extraordinary developer and has made some major contributions to the Z/XDC product.

His first project was adding **TRAP2** support. Now, Z/XDC has support for both 00-opcode type breakpoints and TRAP2 type. The two forms integrate seamlessly with each other, and Z/XDC can be switched back and forth at will to use one or the other as needs arise.

Next, Peter developed our our **dump/XDC** Licensed Feature that permits using Z/XDC to debug IPCS dumps. This was a major development project that Peter tackled with energy and excitement.

Currently (April 2024), Peter is working on adding USS support to Z/XDC. When he's done, Z/XDC will be able to access load modules, program objects and ADATA and DWARF data directly from the USS file system.

Peter lives with his wife in the suburbs of Sydney Australia and communicates with us daily via Zoom.

Peter can be reached at **peter@colesoft.com.**

Calin Cole

Calin (yes, my son) came to ColeSoft in 2013, not as a programmer, but in a far more important capacity. His job was to learn how to run the company, and he has done so brilliantly!

While I may be a pretty decent programmer, I am terrible at running a company! As ColeSoft was growing, that was becoming more and more of a problem. Fortunately, as I was needing a solution, Calin was at a stage in his life where he was ready to step into something new and challenging.

Now, we have more and younger people, enabling us to take on more projects and engage in newer technologies.

Meanwhile, thanks to Calin, I get to spend more of my time puttering around on fill-in projects and pontificating from time to time with historical insights and future directions.

Calin can be reached by at **calin@colesoft.com.**

Other Credits

Z/XDC's support for displaying floating point data in hexadecimal, decimal and binary formats is based on code provided by **David Bond.** At the time he was owner of Tachyon Software (www.tachyonsoft.com) and developer of their Cross Assembler.

Help Whatsnew

(For descriptions of all changes for release V3.1, see the z/XDC® V3.1 Maintenance and New Functiona