

Good for Life: Rules for Writing Quality, Maintainable Assembler Code

By David Cole and Peg Ralph

GOOD code enjoys a good quality of life. It lives longer, requires less care and feeding, and costs less to keep up. But writing good code takes some effort. It requires that a programmer develop a well thought out set of rules for coding. Programmers who consistently apply such rules build more reliable and maintainable products.

Standards applied during the development of a commercial software product determine the cost of its upkeep and its longevity. Generally, the costs of maintenance and upgrades during the life of a successful product will exceed its development costs, and it is likely that the programmers who maintain and upgrade the code will not be the same programmers who developed it. So product code that is so poorly or obscurely written that only its developers can understand it will be impossible to maintain by future developers. This can result in a company having to abandon a product all together. But when programmers write clear, maintainable Assembler code at the onset, they will reduce costs, and extend a product's lifetime.

When developing any program, there are many right ways and numerous wrong ways to write code. Programmers who write good code are those who have learned from experience the importance of developing good personal standards for coding. I've been writing commercial code for almost forty years, and the standards I will present here are ones that have served me well in the program development process.

FIGURE 1: ROUTINE HEADER COMMENTARY

```
TITLE 'RBTOREGS - Obtain All Register Sets Associated with a Given RB'
***** 03/05 Y12
*
* RBTOREGS -- This routine obtains all register sets 03/05 Y12
* associated with a given RB. This includes the following: 03/05 Y12
* - General registers (from the next newer RB/XSB). 03/05 Y12
* - Access registers (from the next newer XSB). 03/05 Y12
* - Floating-point registers (from the owning TCB/STCB). 03/05 Y12
* - Control registers (reconstructed from data found both 03/05 Y12
* in the current RB/XSB and the owning TCB/STCB). 03/05 Y12
*
* ENVIRONMENT: 03/05 Y12
* - R8 (PGMBASE1) = @'#RBTOREG. 03/05 Y12
* - R9 (TWAREG) = @'TWARBRG. 03/05 Y12
* - R11 (GBLREG) = @'YBC-GBL. 03/05 Y12
* - R12 (BVTREG) = @'YBC-BVT. 03/05 Y12
* - R13 (LCLREG) = @'YBC-LCL. 03/05 Y12
*
* INPUTS: 03/05 Y12
* - R1 = @'RBREGS, the output sink for all of the 03/05 Y12
* register sets to be extracted. 03/05 Y12
* - ADDR_ADALAS points to the RB whose registers are to 03/05 Y12
* obtained. 03/05 Y12
* - It has not yet been verified that the location 03/05 Y12
* pointed to by ADDR_ADALAS is that of an RB. 03/05 Y12
*
* RETURN VIA IFRETURN: 03/05 Y12
* - A hard error has occurred. 03/05 Y12
* - R15 = @'Error xdoor. 03/05 Y12
* - All other registers are restored. 03/05 Y12
*
* RETURN VIA IFRETO4: 03/05 Y12
* - An error has occurred that is probably the result of 03/05 Y12
* system control block structure shape changes. 03/05 Y12
* - R15 = @'Error xdoor. 03/05 Y12
* - All other registers are restored. 03/05 Y12
*
***** 03/05 Y12
```

Briefly, the rules I regularly apply when coding symbols appropriately, and using sanity checks include commenting comprehensively, using both at assembly time and at execution time.

COMMENT THOROUGHLY

Thorough commentary is essential to good Assembler coding. I consider commentary to be anything that aids in the understanding and memory of a program's function and purpose. Over the years, I have discovered that human memory deteriorates with age but commentary does not. Overall, when commenting, a programmer should be comprehensive, organized, and consistent.

For commentary purposes, I view a program at three levels of abstraction: routines, units of logic, and individual thoughts.

ROUTINES

A routine (a subroutine for example) is a sequence of code that performs a major or independent chunk of work. Each routine should have comprehensive header commentary. The header should make it unnecessary to read the routine's code to determine either how to use it or what its actions and effects are upon the program. As shown in FIGURE 1, the header commentary should describe the routine's purpose, environment, inputs, outputs, and completion conditions.

UNITS OF LOGIC

A unit of logic is a collection of statements used for some basic purpose. This is a subjective concept that falls somewhere between a routine and a thought. Every unit of logic should have a header describing what the programmer intends that piece of code to accomplish.

As shown in FIGURE 2, this header commentary should state the general purpose of the logic and any special information that may not be obvious to another programmer when scanning the code.

INDIVIDUAL THOUGHTS

A thought is a handful of statements intended to accomplish some elemental purpose. A thought does not need to have its own header but its statements should have enough commentary for other programmers to understand its intent. I recommend separating thoughts from each other with white space as shown in FIGURE 3.

USE SYMBOLS APPROPRIATELY

Methodical use of symbols makes a program's code easier to understand and easier

FIGURE 2: UNIT OF LOGIC COMMENTARY

```
***** 03/05 Y12
* Extract the SCB under which I am running. Make sure it's * 03/05 Y12
* not an ESTAI, and save the owning RB address. * 03/05 Y12
***** 03/05 Y12
      BASR R4,0          PRIME W/BOGUS VALUE      03/05 Y12
      L R15,DBCSCBA     @'CLIENT'S ACTIVE SCB    03/05 Y12
      LA R0,SCBLEN      L'SAME                   03/05 Y12
      BAS R14,READM31    EXTRACT YBC'S RB        02/05 Y12
      B ATWNSCBZ        +0 ERRORS: FORGET IT     03/05 Y12
      B ATWNSCBZ        +4 OC4: FORGET IT        03/05 Y12
      L R15,STORBLKA    +8 AOK: @'EXTRACTED DATA 03/05 Y12
      USING SCB,R15     SCB ITS BASE             03/05 Y12
```

FIGURE 3: THOUGHT COMMENTARY

```
      SLR R4,R4          CLEAR FOR INSERT          03/05 Y12
      ICM R4,7,SCBOWNRA @'OWNING RB              03/05 Y12
      DROP R15          DONE W/SCB BASE          03/05 Y12
      TWNSCBZ DS OH     03/05 Y12

      LA R15,1          "EXTRACT THE OLDEST RB".  03/05 Y12
      *                (IT ACTUALLY DOESN'T      03/05 Y12
      *                MATTER WHICH RB.)         03/05 Y12
      L R1,UHTCB        @'HOME TCB              02/05 Y12
      BAS R14,DRBSRCH   CALL RB-SEARCH          03/05 Y12
      B ATWCHECK        +0 ERROR. PUNT           03/05 Y12
```

FIGURE 4: POOR USE OF SYMBOLS

```
*****
* Construct a dataset name *
*****
LOGNAME DS CL44          DSNAME
        DS AL1          |
TYPENAME DS CL5          V FILE'S TYPE NAME
*
...
MVC LOGNAME,=CL44' '    CLEAR THE BUFFER
MVC LOGNAME(5),TYPENAME FILE'S TYPE NAME
SLR R1,R1              CLEAR FOR INSERT
IC R1,TYPENAME-1      L'TYPE NAME
LA R1,LOGNAME(R1)     Z'PREFIX
MVI 0(R1),C'.'        INSERT A DOT
L R15,X'21C'          @'CURRENT TCB
L R15,12(,R15)        @'TIOT
MVC 1(8,R1),0(R15)    APPEND THE JOBNAME
LA R1,9(,R1)          SCAN -
BASR R14,0            FOR THE -
BCTR R1,0             END OF -
TM 0(R1),255-C' '    THE -
BZR R14              JOBNAME.
MVC 1(20,R1),=C'.GENERAL.PROCESS.LOG' APPEND TEXT
#TEST SIZE=(5+1+8+20,LE,44) FIT CHECK
DC 0Y(X'7FFF'-(44-(5+1+8+20)))
```

to maintain. Programmers should code symbols with discipline and use them comprehensively throughout their programs. If you are not using symbols wherever possible, your code will be difficult for other programmers to comprehend and to modify when future needs arise.

Symbols are more readable if their names have relevance to the program and are not visually similar (example: **ITSOLD** vs. **ITSOLD**). Symbols used appropriately should

document the relationships between the various elements of a program.

FIGURE 4 and FIGURE 5 show samples of code that construct a dataset name. FIGURE 4 shows poor use of symbols. FIGURE 5 shows good use of symbols.

POOR USE OF SYMBOLS

The poor use of symbols in FIGURE 4 would cause several difficulties for someone modifying

or updating the code. For example, in this figure the programmer has used constants to represent the lengths of dataset names (44), job names (8), and the **TYPENAME** field (5). As a result, any programmer responsible for updating these lengths would have to make dependant changes in several places.

In addition, this sample has hardcoded off-sets that refer to the fields in the system control blocks, Prefix Save Area (**PSA**), Task Control Block (**TCB**), and the Task Input Output Table (**TIOT**). Another programmer reading the program would have difficulty understanding the original programmer's intentions. For example, in the "**MVC LOGNAME(5),TYPE-NAME**" statement, was it the programmer's intention to move all of the **TYPENAME** field or just part of it? And in the "**L-L-MVC**" sequence, what **PSA**, **TCB**, and **TIOT** fields is the programmer referencing?

GOOD USE OF SYMBOLS

In **FIGURE 5**, I have used symbols such as equates, references to field lengths, and field names to replace raw numbers as much as possible in order to make the code easier to understand and easier to modify in the future. Doing this reduces the number of changes required when updating a program. For example, a **DSNAME** buffer has to be 44 bytes long. There is no way around this because 44 is the maximum length that a dataset name can be in **MVS**. However, if you must use a number like this, use it only once. Use the number to create an equate and then use the equate everywhere else in the program that the value is needed. That way, if the number changes in the future, the person updating the program will only have to change it once.

I also replaced the hardcoded field lengths in this code sample with references to the length attributes of certain fields. For example, I've changed the length of the jobname from a hardcoded "8" to a reference to the length of the **TIOCJOB** field of the **TIOT** (the system control block field from which the program is obtaining the jobname).

In addition, I changed the references to the length of the **TYPENAME** field from a hardcoded "5" to a reference to the **TYPENAME** field's length attribute. Now, if the length of the **TYPENAME** field is ever changed, any programmer updating the program will not have to change all the instructions and other data that have dependencies upon that length. The assembler will do that automatically.

FIGURE 5: GOOD USE OF SYMBOLS

```

DSNAMEL EQU 44
*
*****
* Construct a dataset name
*****
LOGNAME DS CL(DSNAMEL) DSNAME
DS AL1 |
TYPENAME DS CL5 V FILE'S TYPE NAME
*
...
MVC LOGNAME,=CL(L'LOGNAME)' ' CLEAR THE BUFFER
MVC LOGNAME(L'TYPENAME),TYPENAME FILE'S TYPE NAME
SLR R1,R1 CLEAR FOR INSERT
IC R1,TYPENAME-1 L'TYPE NAME
LA R1,LOGNAME(R1) Z'TYPE NAME
MVI 0(R1),C'.' INSERT A DOT
L R15,PSATOLD-PSA @'CURRENT TCB
L R15,TCBTIO-TCB(,R15) @'TIOT
USING TIOT,R15 DCL BASE FOR TIOT HEADER
MVC 1(L'TIOCJOB,R1),TIOCJOB APPEND THE JOBNAME
DROP R15 DONE W/TIOT HEADER
LA R1,1+L'TIOCJOB(,R1) SCAN -
BASR R14,0 FOR THE -
BCTR R1,0 END OF -
TM 0(R1),255-C' ' THE -
BZR R14 JOBNAME.
MVC 1(20,R1),=C'.GENERAL.PROCESS.LOG' APPEND TEXT
#TEST SIZE=(L'TYPENAME+1+L'TIOCJOB+20,LE,L'LOGNAME)
DC 0Y(X'7FFF'-(L'LOGNAME-(L'TYPENAME+1+L'TIOCJOB+20)))
*
...

```

FIGURE 6: USING A Y-CON TO DOCUMENT IMPLIED REGISTERS

```

TRT whatever
DC 0Y(R1,R2) HIT POINTER AND HIT CODE

```

FIGURE 7: USING Y-CONS TO DOCUMENT THE TARGETS OF A STM

```

STM R3,R5,FSTART
DC 0Y(R3,L'FSTART) SAVE @'FIELD
DC 0Y(R4,L'FEND) SAVE Z'FIELD*
DC 0Y(R5,L'FLEN) SAVE L'FIELD

```

*In my commentary, I use "Z'---" to mean "end of". "@'---" means "address of".

FIGURE 8: USING Y-CONS TO DOCUMENT OTHER IMPLIED REFERENCES

```

MVCSK HERE,THERE
DC 0Y(R0) LENGTH
DC 0Y(R1) SOURCE KEY

SCKPF ,
DC 0Y(R0)

CSP R4,R7
DC 0Y(R5)

STFL 0
DC 0Y(L'FLCACL) PSA+X'C8'

STSI SYSIB
DC 0Y(R0) FUNCTION CODE
DC 0Y(R1) SUBCODE

```

Finally, I changed the references to fields these offsets will ever change, but to document that these are the referenced control blocks and fields.

SANITY CHECKS AND REFERENCE ALERTS USING Y-CONS

A Y-CON (Y-type address constant, example: **DC Y(expression)**) designates a two-byte address field in storage. This instruction is similar to an H-CON but with two big advantages. First, its operand can be an expression instead of just a value. Second, the assembler not only imposes a limit on the expression's resolved value, but also generates a syntax error when the value exceeds that limit. These two characteristics make Y-CONS useful for documenting "hidden" relationships in Assembler code, such as implied register usages, unconventional register requirements, and violated dependencies.

USING Y-CONS TO DOCUMENT IMPLIED REFERENCES

Some instructions, like Translate and Test (**TRT**), imply the use of certain registers (in this case R1 and R2) that the instructions do not directly reference within their operands. The assembler does not report such implied registers and a programmer can easily overlook these types of "hidden" references when scanning a program's code. So as shown in FIGURE 6, I always follow a **TRT** instruction with a zero-length Y-CON that references the instruction's implied registers. This makes the implied registers easier to error check, and it documents the referred registers both inline and in the assembler's symbol cross reference.

Programmers can also use Y-CONS for documenting implied references to fields. For instance, the **LOAD MULTIPLE (LM)** and **STORE MULTIPLE (STM)** instructions often refer not just to ranges of registers, but also to ranges of data fields. FIGURE 7 shows how a programmer can use Y-CONS to document all of the registers used by a **STM** and all of the fields altered by the **STM**.

Y-CONS are handy for highlighting implied references within many other instructions.

FIGURE 8 shows several examples of using Y-CONS to highlight hidden references in code. Note particularly the Store Facility List (**STFL**) instruction that uniquely (and strangely) is defined to make an implied reference, not to a register, but to a specific storage location! (Real address X'00000200').

USING Y-CONS TO DOCUMENT UNCONVENTIONAL API REQUIREMENTS

How many programmers know, without consulting an IBM manual, which registers the MVS **SETLOCK** service affects? I know I don't. I

FIGURE 9: USING Y-CONS TO DOCUMENT REGISTERS AFFECTED BY THE SETLOCK SERVICE

```

SETLOCK OBTAIN,TYPE=LOCAL,MODE=UNCOND
+      L      14,PSALITA-PSA(0,0)  LOCK INTERFACE TABLE ADDRESS
+      L      13,576+8(14,0)      LOAD ADDRESS OF LOCK RTN
+      BALR   14,13                BRANCH ENTER LOCK ROUTINE
DC     0Y(R11,R12,R13,R14) XREF ALTERED REGISTERS

```

FIGURE 10: USING Y-CONS TO DOCUMENT REGISTERS AFFECTED BY CROSS MEMORY POST

```

POST (R2),          ECB ADDRESS          12/04 Y11*
      (R3),          POST CODE           12/04 Y11*
      ASCB=(R5),     TARGET A.S.'S ASCB   12/04 Y11*
      ERRET=CVTBRET, NULL ERROR EXIT ROUTINE 12/04 Y11*
      ECBKEY=0,      TARGET ECB'S KEY IS ANY 12/04 Y11*
      MEMREL=NO,     ERRET RUNS IN MASTER A.S. 12/04 Y11*
      LINKAGE=BRANCH BRANCH ENTRY        12/04 Y11
+      LR      10,R3                . SET POST CODE
+      LR      11,R2                . ECB ADDRESS
+      LA      15,X'800'(0,0)
+      SLL    15,20(0)
+      OR      11,15                . SET CROSS ADDR BIT
+      LR      13,R5                . ASCB ADDRESS
+      LA      12,CVTBRET           . ERRET IN R12
+      OR      12,15                . NOT MEMORY RELATED
+      LA      0,16*0              . ECBKEY IN R0
+      OR      10,15                . SET ECBKEY FLAG BIT
+      L       15,16(0,0)
+      L       15,CVTOPT01-CVTMAP(0,15)
+      BALR   14,15                . BRANCH ENTRY
DC     0Y(R0,R1,R2,R3,R4,R5,R6,R7) ONLY R9 SAVED. ALL 03/05 Y12
DC     0Y(R8,R10,R11,R12,R13,R14,R15) OTHERS TRASHED! 03/05 Y12

```

FIGURE 11: ASSEMBLER CONSISTENCY CHECK

```

MYNAME DC C'YBC'
MYNAME1 EQU L'MYNAME'
...
MVC YBCDWORK(MYNAME1),MYNAME BUILD THE DEFAULT - 02/04 Y10
MVC YBCDWORK+MYNAME1(8-MYNAME1),=CL8'SLIST' - 02/04 Y10
* SLIST-LIBRARY DDNAME 02/04 Y10

```

FIGURE 12: DETECTING BUFFER OVERFLOW

```

LEQM MVC YBCMSG(L'YBCMSG),BLANKS CLEAR MSG BUFFER 03/05 Y12
      USING LEQMSG,YBCMSG DCL MSG FIELDS OVERLAY 03/05 Y12
* DC 0YL2(X'7FFF'-(L'YBCMSG-L'LEQMSG)) DOES LEQMSG
      FIT INTO YBCMSG? 03/05 Y12

```

also can't instantly recall which registers the Cross Memory **POST** service saves and which registers it wipes out. However, by using Y-CONS as shown in FIGURE 9 and FIGURE 10, the registers affected by these services are cross-referenced and I can save myself a trip to the books when reviewing code.

USING Y-CONS WITH SANITY CHECKS

A sanity check is any programmatic action taken at assembly time or at execution time that proves an assertion relevant to the structure of a program, the logic of a program, the intent of the code, or state of the current environment. When troubleshooting code, the presence of sanity checks can save time and effort by guiding a programmer away from irrelevant inquiries.

ASSEMBLY TIME SANITY CHECKS

At assembly time, a sanity check is any instruction that leads to a syntax error, should the code not meet a required condition. Programmers can accomplish many sanity checks simply by using symbols so that a violation of a dependency automatically results in a syntax error. For example, as FIGURE 11 shows, if the value of **MYNAME** is 9 or greater the assembler will report a syntax error.

However, many situations arise in which the assembler has no direct ability to detect a problem. For example, a message constructed in a buffer must not overflow that buffer. But if a programmer builds the message from a series of constants and fixed length elements, there will be no easy way for the assembler to check directly for overflow.

However, using Y-CONs, there is an easy way to build an assembler time sanity check that is sensitive to overflow. Simply use the length of the message and the length of the buffer to concoct a constant whose value is syntactically valid if (and only if) the message is not too long. And by making the constant zero-length, you can even place it inline (at the point in the code where the tested condition matters) without interfering with the rest of the code. FIGURE 12 shows an example of a sanity check that uses arithmetic within a Y-CON to produce a legal value if, and only if, the length of **LEQMSG** is less than or equal to the length of **YBCMSG**. (When the condition is violated, that is, when **LEQMSG** is longer than **YBCMSG**, the Y-CON arithmetic will result in a value that exceeds that which is permitted for Y-CONs, causing the assembler to issue a syntax error.)

Since testing statements are generally syntactically complex, I have written a macro (**#TEST'**) that allows me to code the relationship that I want to test, and it takes care of the messy syntax necessary to verify that relationship. For example, in Cole Software's product, I have provided fixed length work areas for the private use of many of its subroutines. Each of these subroutines has a small DSECT map that defines the routine's use of its work area. It is important that the routine's use of the work area not exceed the work area's length. FIGURE 13 shows how I use **#TEST** to check for work area overflow.

I can use **#TESTs** to document logic dependencies of all kinds throughout our product. For example, FIGURE 14 shows how I use **#TEST** to remind me to update a table in our product whenever I change the product's release number variable. And FIGURE 15 shows how I use **#TEST** to document a logic

FIGURE 13: USING #TEST TO DETECT BUFFER OVERFLOW

```
***** 02/04 Y10
* Local RSTKWORK fields * 12/00 S21
***** 12/04 Y11
ATNWORK DSECT , 12/04 Y11
ATWN @ADDRESS ALET,ASID 03/05 Y12
+ATWNALET DC F'0' ALET 03/05 Y12
+ATWNAS_F DC F'0' ASID (FWORD) 03/05 Y12
+ATWNASID EQU *-2,2 ASID (HWORD) 03/05 Y12
+ATWN_ALAS EQU ATWNALET,*-ATWNALET 03/05 Y12
ATWNIRB# DS H IRB'S # 12/04 Y11
ATWNFLGS DS B LOCAL FLAGS 12/04 Y11
#TEST SIZE=(*-ATWNASID,LE,L'RSTKWORK) FIT CHECK 12/04 Y11
+ DC 0YL2(X'7FFF'-(L'RSTKWORK-(*-ATWNASID)))
YBCMISC CSECT , RESUME CODE SECTION 12/04 Y11
...
...
...
USING RSTK,RSTKREG DCL GENERIC RSTK BASE 02/04 Y10
USING ATWNRSTK,RSTKWORK DCL LOCAL WORK FIELDS 12/04 Y11
...

```

FIGURE 14: USING #TEST TO ALERT A RELEASE CHANGE

```
***** 12/94 X31
* The following table is used to determine the release of * 12/94 X31
* older YBC's by their load module lengths. (Sorted) * 06/96 X32
***** 12/94 X31
OLDXDCS DS OF ALIGNMENT
DC X'0000DD68',C'X3.3' X33 10/97 X34
DC X'00010060',C'S1.0' S10 01/99 X35
DC X'00011260',C'S1.1' S11 01/00 S20
DC X'00011928',C'S2.0' S20 08/00 S21
DC X'00018210',C'X2.0' X20 12/94 X31
DC X'000203D8',C'X2.1' X21 12/94 X31
DC X'00035EB8',C'X2.2' X22 12/94 X31
DC X'00036990',C'X3.0' X30 12/94 X31
DC X'0003ADD8',C'X3.1' X31 06/96 X32
DC X'00043928',C'X3.2' X32 04/97 X33
DC X'FFFFFFFF',C'UNKN' DELIMITER 12/94 X31
#TEST SIZE=(&XDVER#,EQ,12) ALERT WHEN RELEASE 03/05 Y12*
CHANGES FROM Y1.2 03/05 Y12
+ DC 0YL2(X'7FFF'-(12)+12,X'7FFF'-(12)+12)

```

FIGURE 15: USING #TEST TO DOCUMENT A LOGIC DEPENDENCY

```
L R1,CVTPTTR --> CVT 08/90 X21
L R1,CVTQLPAQ-CVTMAP(,R1) --> LPA-Q ANCHOR X20
USING CENTRY,R1 DCL CDE BASE X20
#TEST SIZE=(CDCHAIN-CENTRY,EQ,0) (ERROR IF CHAIN FIELD X20*
NOT FIRST) X20
LPQSCAN ICM R1,15,CDCHAIN --> NEXT CDE; ANY MORE? X20

```

FIGURE 16: ABORTIVE TRAP EXAMPLES

```
Executing an EX **+1 causes an 0C6:
LTR R1,R1
BNZ **+8
EX 0,**+1

Executing an invalid opcode causes an 0C1:
LTR R1,R1
BNZ **+6
DC Y(0)

```

dependency in code that requires a CDE's ever change, but it is handy to be alerted to that dependency when reviewing the code).

EXECUTION TIME SANITY CHECKS

At execution time, sanity checks can range from corrective error handlers to abortive program check traps. Corrective error handlers, however, generally are complex to write. Abortive traps, on the other hand, are much simpler to write than error handlers. An abortive trap is an instruction that causes an immediate program check in the event that some condition occurs that a programmer considers highly unlikely to occur. FIGURE 16 shows two examples of abortive traps.

When an error condition is likely to occur, then programmers need to write handlers regardless of the complexity. However, other conditions might be so unlikely that it is not worth the effort of writing a full-blown handler. Instead, coding an abortive trap is trivially easy, so why not do at least that?

I have written a macro (**#DIE**¹) to generate conditional, abortive traps that have explanatory messages associated with them. As FIGURE 17 shows, **#DIE** generates what I call a “DEAD-trap.” Depending upon the operands given, you can generate the DEAD-trap either in-line (with a conditional skip around it) or

FIGURE 17: USING #DIE MACRO TO GENERATE A DEAD-TRAP

Executing a **#DIE** macro permits a condition test and causes a stylized OC1 followed by a message:

```

LTR R1,R1

      #DIE Z,'LOGIC ERROR IF NO PLIST!'
+     BNZ DIE0035Z
+     DC  X'00DEAD',AL1(29)
+     DC  C'0035 LOGIC ERROR IF NO PLIST!'
+DIE0035Z DS  OH
  
```

FIGURE 18: USING #DIE TO ABORT UPON RECURSION STACK UNDERFLOW

```

***** * 10/93 X22
* Vectored return to caller - Decrement recursion depth * 10/93 X22
* gauge, then make the vectored return. * 10/93 X22
***** * 10/93 X22
ADDRRET8 BAS R0,ADDRRET +8 AOK, NORMAL DELIMITER 10/93 X22
ADDRRET4 BAS R0,ADDRRET +4 AOK, SPECIAL DLIMITER 10/93 X22
ADDRRETO BAS R0,ADDRRET +0 ERROR, MSG DOOR RTURN'D 10/93 X22
ADDRRET LA R14,ADDRRET VECTOR BASE 10/93 X22
SLR R14,R0 RETURN OFFSET 10/93 X22

L R0,DARCDPTH LOAD RECURSION DEPTH GAUGE 10/93 X22
BCTR R0,0 DECREMENT 10/93 X22
LTR R0,R0 UNDERFLOW? 10/93 X22
#DIE M, RECURSION UNDERFLOW! YES, LOGIC ERROR 10/93 X22
+ BNM DIE7023Z SKIP IF OK
+ DC X'00DEAD',AL1(25),C'7023 RECURSION UNDERFLOW!' 02/99 XLQ
+DIE7023Z DS OH RECEIVE SKIP AROUND TRAP
  
```

TIRED
of reading
dumps?

Z/XDC
Work smarter, not harder

Debug Assembler
programs in less
time, with less
tedium...

And with less
EYE STRAIN

cole software[®]
Powerful assembler development tools

800-XDC-5150
www.colesoft.com

out-of-line as a literal operand of a branching instruction. If you want, you can even have the DEAD-trap to include a special message. Just provide the message in a quoted string as the macro's last operand. (If you do not include your own message, then the macro generates a default message, a unique 4-digit number).

DEAD-traps are wonderfully useful devices. Some examples of situations where I have used **#DIE** include: catching illogical conditions, documenting interface dependencies, documenting code points for future expansion, verifying complex code, and guarding against future code changes that are incompatible with current logic.

FIGURE 18 is an example of how I use **#DIE** to catch an illogical condition such as a stack underflow. If the developer wrote the program correctly, underflow should never occur. However, a DEAD-trap ensures that you will know if underflow does occur.

DEAD-traps are ideal for documenting interface dependencies. Generally, I use these traps at interfaces to subroutines that I do not fully trust or over which I do not have any control (such as exits into other products).

FIGURE 19 shows my use of **#DIE** to verify a caller's environment.

Programmers can use DEAD-traps to document code points for future expansion and to catch partial implementations. For example, Cole Software's product reads Associated Data (**ADATA**) file from either of two sources, a **//SYSADATA** file, or a Generalized Object File Format (**GOFF**) file. FIGURE 20 shows my use of **#DIE** to document an insertion point for future code to support new types of **ADATA** sources. It will also trap execution if my type flags are not set as expected.

DEAD-traps are good for verifying and documenting complex code. A programmer can use these traps to document and prove subtle or indirect relationships within code. As FIGURE 21 shows, you can use double checking logic and DEAD-traps for code you are uncertain about, either to prove you wrong or to create confidence that you are right.

CONCLUSION

Comprehensive commentary, appropriate use of symbols, use of sanity checks at assembly time and execution time are among the rules I regularly apply when writing code. Consistent application of standards such as these, combined with an overall clarity of thought, will extend the usefulness and reliability of any commercial product written in Assembler code.

FIGURE 19: USING #DIE TO VERIFY A CALLER'S ENVIRONMENT


```
***** * 04/99 S11
* Make sure I'm being called with the right TWA (TWAMAP). * 04/99 S11
***** * 04/99 S11
      CLI  TWATYPE,TWAMAP      "TWAMAP" TWA?      04/99 S11
      #DIE NE                   NO, LOGIC ERROR    04/99 S11
+      BE  DIE5618Z             SKIP IF OK          04/99 S11
+      DC  X'00DEAD',AL1(4),C'5618'                02/99 XLQ
+DIE5618Z DS  OH                   RECEIVE SKIP AROUND TRAP
```

FIGURE 20: USING #DIE TO DOCUMENT AN INSERTION POINT

```
***** * 08/99 S11
* GOFF record processing. * 08/99 S11
***** * 08/99 S11
      SPACE 1
      TM  ADATFLGS,ADFGOFF    GOFF PROVIDER?      08/99 S11
      BZ  SMGOFFZ            NO, SKIP              08/99 S11
      ...
SMGOFFZ DS  OH                08/99 S11
***** * 08/99 S11
* SYSADATA record processing. * 03/00 S20
***** * 08/99 S11
      TM  ADATFLGS,ADFSYSAD  SYSADATA PROVIDER?  08/99 S11
      BZ  SMSADATAZ          NO, SKIP              03/00 S20
      ...
SMSADATAZ DS  OH                03/00 S20
***** * 08/99 S11
* Unknown record processing. Development error. * 08/99 S11
***** * 08/99 S11
      #DIE ,
+      NOP DIE5622Z           GATE                  08/99 S11
+      DC  X'00DEAD',AL1(4),C'5622'                02/99 XLQ
+DIE5622Z DS  OH                   RECEIVE SKIP AROUND TRAP
```

FIGURE 21: USING #DIE TO DOCUMENT A LOGIC ERROR

```
***** * 08/99 S11
* The new ADATA record will not fit into the current cache * 08/99 S11
* block, so I'm going to have to get a new block. But * 08/99 S11
* first, I need to freemain the residue (if any) from the * 08/99 S11
* current cache block. * 08/99 S11
***** * 08/99 S11
      ...
*      SR  R0,R1                L'RESIDUE TO FREEMAIN; 08/99 S11
*                                ANY?                08/99 S11
*                                NO, SKIP THE FREEMAIN 08/99 S11
*                                (LOGIC ERROR)        08/99 S11
+      #DIE M
+      BNM DIE5626Z           SKIP IF OK          02/99 XLQ
+      DC  X'00DEAD',AL1(4),C'5626'                02/99 XLQ
+DIE5626Z DS  OH RECEIVE           SKIP AROUND TRAP
```


It may take extra development time but in the long-term, you will reduce the cost of a product's upkeep and increase its longevity. 

NaSPA member Peg Ralph is Cole Software's resident technical writer and Web master.

*Note: The #TEST and #DIE macros are freely available for download from Cole Software's Web site at: <http://www.cole-software.com/utilities.html>.

NaSPA member David B. Cole has nearly 40 years of experience in MVS (and predecessor) systems programming, with concentration in Assembler and MVS internals. He has been the president of Cole Software LLC, a small software development company located in Afton, Virginia, since 1988. Dave is also the lead developer of Cole Software's core product, the source level assembler debugger, z/XDC®.

©2005 Technical Enterprises, Inc. Reprinted with permission from **Technical Support** magazine. For subscription information go to www.naspa.com, email mbrship@naspa.com or call 414-908-4945, Ext. 115.



Z/XDC[®]

Work smarter, not harder

Does chasing
bugs make you
DOG-TIRED?

Debug your
Assembler in
less time,
with less
effort...

And with less

SWEAT

cole software[®]

Powerful assembler development tools

800-XDC-5150

www.colesoft.com