Considerate Programming
Reducing the Maintenance Costs of Commercial Quality Code

SHARE, August 2003
Session 8139
(Tuesday, 9:30AM)

Dave Cole
President, Cole Software, LLC
dbcole@colesoft.com

(Generated July 29, 2003)

(Generated July 29, 2003)

Contents

Figures

Considerate Programming
Reducing the Maintenance Costs of Commercial Quality Code

In programming, there are many right ways, and many many more wrong ways, to write code. As a programmer gains experience, he/she gradually develops a personal set of rules to aid in the choices he must make when writing code. I refer to these rules as an "aesthetic". When coding choices are guided by a clear and well thought out aesthetic, the resulting product will be more reliable and more maintainable.

The Development of an Aesthetic

I remember that when I first began to write assembler code, I felt somewhat overwhelmed, not just by the number of books I had to comprehend (somewhere around 10 or so at the time), but also by the shear massiveness of the variety of ways there are to write code.

The first language I learned was Fortran. It was simple and straightforward. The language syntax was pretty strict. (One statement per card. Required column alignments, etc.) And coding choices were pretty simple. But writing commentary was rather inconvenient in that language, and doing so was a pain in the butt anyway. (Too much thinking! Too much typing! Too much time wasted!)

The second language I learned was Algol (Algol-60). It had a couple of features that I had never seen before. Commentary on every statement, for one (yawn). And it would allow me to pack as many language statements per card as I could fit. (Wow!) And I could use abbreviations. (Double WOW!) I could make my program really really small! (Yeah, I had my confusions.)

And so I wrote this 200 statement program, and crammed it onto around 15 or 20 cards or so. Boy was I happy. Happy until I had to reread it a couple of weeks later to try to figure out what I was trying to do. Happy until I found a bug and had to insert a change! Easy to do with an editor. Not so easy to do with a keypunch. (Hold down DUP up to the insert point, press your thumb hard against the old card to keep it from advancing, type the insertion into the new card while still holding the old card without wiggling it, DUP the rest of the card. Remember what has spilled from the new card and type it onto the next card. Rinse and repeat.) Editors did not exist back then. Life was primitive!

Cramming all statements into the highest density possible was not a good choice for writing code. I found this out then for source code. I found it out later for object code. Density and clarity tend to have an inverse relationship to each other.

When I started programming, machines were flashy but slow. Huge but utterly tiny! Back then, an awesomely "large" machine would fill a room and have a whole megabyte(!) of real[1] storage to wallow in. So early in my programming life, sure, "getting it to work" was a high priority. But getting it to work in interesting and tricky ways was an even higher priority. And getting it to work as fast as possible in as few bytes as possible, that was tops! Making it comprehensible to other people? Irrelevant! In fact I gave myself bonus points if the program worked, worked fast as hell, and people had to ask me how it worked. That was an aesthetic, not a good one, but an aesthetic nonetheless.

Over time, both internal realizations and external priorities changed. As regions and private areas increased in size, the importance of writing small tight code dwindled eventually to an irrelevancy. As hardware speeds kept doubling and doubling again, writing efficient code still retained its place, but for most of my work, it became less and less important as well. Writing conceptually clean/efficient code still is important to me, but squeezing that last possible cycle out of a routine? I'll leave that to others.

And as I came to deal with more and more existing code, written both by myself and by others, I came to value more and more the importance of that lowly comment statement.

So as my early aesthetics declined, others have ascended to take their place: Chief among these is "clarity". Clarity of code, clarity of commentary, clarity of language, and (where possible) clarity of thought. In this paper, I will touch upon some of the styles and techniques that I have developed that aid me in the programming process.

A lot of the suggestions that follow are "rules" that I try to follow in my own coding. They work very well for me, and I think that they will work quite well for anyone who has the interest and the patience to adopt them. But please do not think that I consider these rules to be complete or even comprehensive. They are not. Please do not think that I consider

[1] The word "virtual" had not yet been invented.

them to be the only right way to code. I do not. I do think that the basic ideas that I'm about to describe are of considerable benefit, and if you agree with even some of them, I will be pleased, but I do not think that my ways of implementing these ideas are the only good choices.

In this paper, I will use real code samples to illustrate the various ideas that I'm about to put forth. These samples have been extracted from my current development level of the upcoming z/XDC™, Cole Software's primary product.

Commercial Quality Code - Premises

I have been writing commercial code for around 27 years or so. During that time, I have developed my premises and prejudices. Here's a list. Like mathematics, whether or not you agree with these premises is relevant only in the sense that if you chose different premises, you will derive a different aesthetic. That's fine, but outside the scope of this paper.

Premises:

- A commercial product is intended to have a long lifetime.
- The product should, within reason, be as bug free as possible.
- The vendor will accept bug reports and will correct reported bugs.
- The product will be enhanced/upgraded from time to time.
- Maintenance/upgrade costs, over time, will exceed development costs.
- Maintenance programmers (and even upgrade programmers) probably will not be the same people as the development programmers.
- Over time, the original developers will leave the product.
- As the number of programmers involved in a project increases, the abilities of the programmers will tend to drop (rise?) towards an average.
- Code written by “superior” programmers, sooner or later is going to have to be maintained by “average” programmers.
- Code that can be understood only by “superior” programmers is not feasibly maintainable and may even have to be abandoned.
- Code that can be understood only by “superior” programmers is probably badly written in the first place. (Certainly, it is badly commented!)
- Complex programming requires good memory. Memory deteriorates with age. Commentary doesn't.
- As a “superior” programmer ages, he either will trend towards the average or will die or will retire (either to pasture or to management).

Techniques:

Ok, here's the motherhood and apple pie stuff. When writing commercial quality code:

- Document thoroughly.
- Use a consistent style.
- Use clear, obvious logic. Explain subtleties.
- Reduce the complexity of comprehension.
- Write code that is linear. Avoid convoluted logic.
- Be cautious and conservative.
- Be defensive. Be careful about trusting your environment.
- Write self-adjusting code.
- Use coding techniques to detect errors early.
- Minimize the use of magic numbers.

Documentation

Thorough documentation cannot be overstressed! Without it, it will cost more for successor programmers to understand, fix, and upgrade your code. Keep in mind that a "successor programmer" might even be yourself, a year later, or a decade later, or a lifetime later. Trust me: Anything you wrote a decade or three ago will be as new and wondrous to you as Swahili. So be considerate of your successor. You never know who it might be.

Documentation can take many forms: Commentary, symbol usage, assembly time sanity checks, execution time sanity checks. In other words, documentation is anything that aids in a human's understanding of the program. This paper will discuss these forms in detail.

Commentary

Commentary should be comprehensive, organized, consistently designed, methodically applied, and neat in appearance. White space should be used, both in commentary and in code, to help the reader find the start and end of each thought, idea, or other boundary.

For the purposes of commentary, I consider code to be broken down into three levels:

- Subroutines,
- Units of logic, and
- Thoughts.

Subroutines consist of several units of logic. Units of logic consist of only a handful of thoughts. And thoughts consist of only a handful of statements.

Write commentary blocks at the heads of subroutines and units of logic. Insert white space between individual thoughts. Include enough statement commentary for the thoughts to be understood.

Subroutine Header Commentary

Subroutine interfaces need to be thoroughly documented. Header commentary should be written to describe the subroutine's purpose, its environment, its inputs, its outputs, and its various return conditions. It should not be necessary to read the subroutine's code to determine either how to use it or what its actions and effects are upon the environment. Figure **1** shows an example of header commentary.

```
         TITLE 'RBTOREGS -- Obtain All Register Sets Associated with a *
               Given RB'
************************************************************* 05/02 Z12
*                                                           * 05/02 Z12
* RBTOREGS -- This routine obtains all register sets        * 05/02 Z12
* associated with a given RB. This includes the following:  * 05/02 Z12
*   - General registers (from the next newer RB/XSB).       * 05/02 Z12
*   - Access registers (from the next newer XSB).           * 05/02 Z12
*   - Floating-point registers (from the owning TCB/STCB).  * 05/02 Z12
*   - Control registers (reconstructed from data found both * 05/02 Z12
*     in the current RB/XSB and the owning TCB/STCB).       * 05/02 Z12
*                                                           * 05/02 Z12
* ENVIRONMENT:                                              * 05/02 Z12
*   - R8 (PGMBASE1) = @'#RBTOREG.                           * 05/02 Z12
*   - R9 (TWAREG)   = @'TWARBRG.                            * 05/02 Z12
*   - R11 (GBLREG)  = @'XDC-GBL.                            * 05/02 Z12
*   - R12 (BVTREG)  = @'XDC-BVT.                            * 05/02 Z12
*   - R13 (LCLREG)  = @'XDC-LCL.                            * 05/02 Z12
*                                                           * 05/02 Z12
* INPUTS:                                                   * 05/02 Z12
*   - R1 = @'RBREGS, the output sink for all of the         * 05/02 Z12
*     register sets to be extracted.                        * 05/02 Z12
*   - ADDR_ADALAS points to the RB whose registers are to   * 05/02 Z12
*     obtained.                                             * 05/02 Z12
*   - It has not yet been verified that the location        * 05/02 Z12
*     pointed to by ADDR_ADALAS is that of an RB.           * 05/02 Z12
*                                                           * 05/02 Z12
* RETURN TO 0(,R14):                                        * 05/02 Z12
*   - A hard error has occurred.                            * 05/02 Z12
*   - R15 = @'Error xdoor.                                  * 05/02 Z12
*   - All other registers are restored.                     * 05/02 Z12
*                                                           * 05/02 Z12
* RETURN T0 4(,R14):                                        * 05/02 Z12
*   - An error has occurred that is probably the result of  * 05/02 Z12
*     system control block structure shape changes.         * 05/02 Z12
*   - R15 = @'Error xdoor.                                  * 05/02 Z12
*   - All other registers are restored.                     * 05/02 Z12
*                                                           * 05/02 Z12
* RETURN TO 8(,R14):                                        * 05/02 Z12
*   - The desired registers have been extracted into        * 05/02 Z12
*     RBREGS.                                               * 05/02 Z12
*   - All registers are restored.                           * 05/02 Z12
*                                                           * 05/02 Z12
************************************************************* 05/02 Z12
```

1 Subroutine header commentary

Unit of Logic Commentary

A “unit of logic” consists of the collection of statements needed to accomplish some elemental purpose. This is a real subjective concept that falls somewhere between a subroutine and a thought. A unit of logic generally consists of not more than a handful of thoughts.

```
************************************************************ 12/00 S21
* Extract the SCB under which I am running. Make sure it's  * 12/00 S21
* not an ESTAI, and save the owning RB address.             * 12/00 S21
************************************************************ 12/00 S21

         BASR  R4,0                PRIME W/BOGUS VALUE        12/00 S21
         L     R15,DBCSCBA         @'CLIENT'S ACTIVE SCB      12/00 S21
         LA    R0,SCBLEN           L'SAME                     12/00 S21
         BAS   R14,READM31         EXTRACT XDC'S RB           06/02 Z12
         B     ATWNSCBZ            +0 ERRORS: FORGET IT       12/00 S21
         B     ATWNSCBZ            +4 0C4: FORGET IT          12/00 S21
         L     R15,STORBLKA        +8 AOK: @'EXTRACTED DATA   12/00 S21
         USING SCB,R15             SCB ITS BASE               12/00 S21

         TM    SCBFLGS1,SCBSTAI    (E)STAI?                   12/00 S21
         BNZ   ATWNSCBZ            YES, ENVIRONMENT'S NOT     12/00 S21
*                                  RIGHT                      12/00 S21
         SLR   R4,R4               CLEAR FOR INSERT           12/00 S21
         ICM   R4,7,SCBOWNRA       @'OWNING RB                12/00 S21
         DROP  R15                 DONE W/SCB BASE            12/00 S21
ATWNSCBZ DS    0H                                            12/00 S21


************************************************************ 12/00 S21
* I need to scan the current RB queue. Start by determining * 12/00 S21
* the number of RBs queued.                                 * 12/00 S21
************************************************************ 12/00 S21

         LA    R15,1               "EXTRACT THE OLDEST RB".   12/00 S21
*                                  (IT ACTUALLY DOESN'T       12/00 S21
*                                  MATTER WHICH RB.)          12/00 S21
         L     R1,UHTCB            @'HOME TCB                 06/02 Z12
         BAS   R14,DRBSRCH         CALL RB-SEARCH             12/00 S21
         B     ATWCHECK            +0 ERROR. PUNT             12/00 S21
         B     ATWCHECK            +4 REQUESTED RB# (1) IS    12/00 S21
*                                     TOO HIGH (NOT BLOODY    12/00 S21
*                                     LIKELY)                 12/00 S21
         LR    R3,R0               +8 AOK; SAVE QUEUED RB     12/00 S21
```

2 Commenting units of logic

Every unit of logic should have a header comment block describing specifically what the following code is intended to accomplish. This commentary should give the general purpose of the logic as well as any special information that is not obvious to the casual peruser of the code. Figure **2** shows an example of unit of logic commentary.

A thought does not have its own commentary header, but its statements should have enough commentary for the thought to be understood. Thoughts should be separated from each other by white space. (See figure **2**.)

Code Signatures

In the several illustrations used within this paper, you may have noticed a heavy usage of what I call code signatures. These are stylized strings located at the righthand end of the commentary fields (**`05/02 Z12`**, for example). They are brief bits of information that serve two purposes: First, they indicate that programming code has been changed. Second,they roughly describe the why, when, and/or who of a change.

```
         MACRO
         DCBD  &DSORG=,&DEVD=
         ...
.*************************************************************************
.*                                                                       *
.*       ...                                                             *
.*                                                                       *
.*             DFSMS/MVS 1.4.0 CHANGES                             @LAA
.*$LA=SERVICE,HDZ11D0,960205,SJPLGEC: SERVICEABILITY CHANGES       @LAA
.*$LB=GT32K,HDZ11F0,980317,SJPLRK:  >32K TAPE BLOCKSIZE           @LBA
.*                                                                       *
.*       ...                                                             *
.*                                                                       *
.*************************************************************************
.*
         ...
DCBOFUEX EQU   DCBBIT6        SET TO 0 BY AN I/O SUPPORT FUNCTION WHEN
*                             THAT FUNCTION TAKES A USER EXIT. SET TO 1
*                             ON RETURN FROM USER EXIT TO THE I/O
*                             SUPPORT FUNCTION WHICH TOOK THE EXIT.
DCBOLOCK EQU   DCBOFUEX       SAME USE AS DCBOFUEX                 @LAA
DCBOFIOF EQU   DCBBIT7        SET TO 1 BY AN I/O SUPPORT FUNCTION IF
*                             DCB IS TO BE PROCESSED BY THAT FUNCTION
DCBOBUSY EQU   DCBOFIOF       SAME USE AS DCBOFIOF                 @LAA
         ...
```

3 Typical IBM style code signature

IBM uses code signatures in many ways. One is as links to commentary that describes the reason and intent of the change. (See figure **3**.)

```
*************************************************************
*        FOUND SEGMENT OF REPEATING TEXT (5 OR MORE         *
*        CONSECUTIVE OCCURRENCES OF THE SAME BYTE).         *
*        PLACE RA ORDER WITHIN SCREEN IMAGE.                *
*************************************************************
         SPACE 1
ORDER460 MVC   1+2(1,R4),0(R3)     COPY CHARACTER TO REPEAT   03/00 S20
         LA    R3,0(R14,R3)        --> NEXT SOURCE BYTE TO SCAN
         SPACE 1                                              03/00 S20
         LR    R15,R1              COPY SCREEN OFFSET         S20-0111H
         XR    R14,R14             CLEAR FOR DIVIDE           S20-0111H
         LH    R0,PNDGCOLS         #'COLUMNS IN THE PENDING   S20-0111H
*                                  SCREEN                     S20-0111H
         DR    R14,R0              CONVERT THE SCREEN OFFSET  S20-0111H
         DC    0Y(R14)             - TO A COLUMN OFFSET       S20-0111H
         DC    0Y(R15)             - AND A ROW OFFSET         S20-0111H
         LA    R1,1(,R14)          COLUMN POSITION            S20-0111H
         LA    R15,1(,R15)         ROW POSITION               S20-0111H
         SLL   R15,8               ALIGN                      S20-0111H
         ALR   R1,R15              COMBINE: 0000rrcc (NOTE,   S20-0111H
*                                  CARRY OF cc INTO rr OR OF  S20-0111H
*                                  rr INTO 00 IS POSSIBLE AND S20-0111H
*                                  DEALT WITH IN THE ROWCOL   S20-0111H
*                                  ROUTINE.)                  S20-0111H
         SPACE 1                                              03/00 S20
         OI    TSOFLAG1,TSO1WRAP   PERMIT SBA RESOLUTION TO   12/99 S11
*                                  WRAP.                      12/99 S11
         @CALL SBA                 CONVERT RRCC TO SBA ORDER
         DC    0Y(R15,R0,R1)       XREF ALTERED REGISTERS     03/00 S20
         #DIE  C,15                +0 BAD ROW/COL; LOGIC ERR  12/99 JDS
*        FALL  THRU                +4 GOOD ROW/COL            03/00 S20
```

4 Code signatures used at Cole Software

At Cole Software, we use either of two styles of code signatures. (see figure **4**.)

! S20-0111H
This style is used for the source code changes corresponding to a maintenance fix. It serves to document that the fix has, in fact, been propagated to the product's source. “S20-0111H” is the name of the fix. This name indicates that the fix was applied to release S2.0 of XDC/SE®, and that it was the 8th fix (“H”) written in November 2001.

! 03/00 S20
06/02 FHC
This style documents the date when a code change was made (March 2000, in the first case). It also shows either the product's version and release for which the change was made (XDC/SE® S2.0) or the person who made the change (Frank H. Chu).

Other Commentary

If you use subroutines that have vectored returns, then use commentary to document the possible return points. Figure **5** shows an example of this.

```
          BAS   R14,DRBSRCH          EXTRACT THE RB; AOK?       12/00 S21
          B     ATWCHECK             +0 ERROR                   12/00 S21
          B     ATWCHECK             +4 REQUESTED RB# NOT THERE 12/00 S21
*                                        (AGAIN, NOT LIKELY)    12/00 S21
******** FALL  THRU                 +8 GOT IT.                 12/00 S21
```

5 Vectored returns: Lead the comments with +0 +4 +8 etc.

If you have code that is dependent upon a series of fields being adjacent, then document that dependency by drawing a vertical arrow in the field commentary. (See figure **6**.)

```
          #LSSEPC DOC=,LIST=
+LSSEPC   DSECT ,                                          04/93 X22
+LSPEGRGS DS    XL(R#*RW)          GENERAL REGISTERS 0-15  06/01 S21
+LSPEARGS DS    XL(AR#*ARW)        ACCESS REGISTERS 0-15   06/01 S21
+LSPEPKM  DS    XL2              | PSW KEY MASK            04/93 X22
+LSPESASN DS    H                | SECONDARY A.S. NUMBER   04/93 X22
+LSPEEAX  DS    XL2              | EXTENDED AUTHORITY INDEX 04/93 X22
+LSPEPASN DS    H                | PRIMARY A.S. NUMBER     04/93 X22
+LSPEPSW  DS    XL8              V PSW FOR USE BY PR       04/93 X22
+         DS    XL4                RESERVED                04/93 X22
          ...
          LM    R3,R4,LSPEPKM      LOAD PKM SASN EAX & PASN  02/02 Z12
          DC    0Y(R3,L'LSPEPKM,L'LSPESASN) (DOC IMPLIED REF)* 02/03 Z12
          DC    0Y(R3,L'LSPEEAX,L'LSPEPASN) (DOC IMPLIED REF)  02/03 Z12
```

(*Patience. These zero-length Y-CONs are discussed later in this paper.)

6 Documenting adjacent fields dependencies

If you make complex macro calls, consider documenting each operand separately. Figure **7** shows a pretty complicated use of the POST macro. The comments make it easier to understand.

```
          POST  (R2),              ECB ADDRESS                01/94 X22*
                (R3),              POST CODE                  04/93 X22*
                ASCB=(R5),         TARGET SPACE'S ASCB        04/93 X22*
                ERRET=CVTBRET,     NULL ERROR EXIT ROUTINE    04/93 X22*
                ECBKEY=0,          TARGET ECB'S KEY IS ANY    04/93 X22*
                MEMREL=NO,         ERRET RUNS IN MASTER A.S.  04/93 X22*
                LINKAGE=BRANCH     BRANCH ENTRY               04/93 X22
```

7 Documenting individual macro operands

Symbol Usage

Proper symbol usage is a very important part of documentation. Of course, choose symbol names that have relevance[1]. Avoid using symbols that are visually similar[2].

[1] I'd like to strangle the guy who would define "**R2 EQU 0**"!

[2] **DSTHWQE** vs. **DSHTWQE**, for example. And **ITSOLD** vs. **ITS0LD**.

But more than that, comprehensive use of symbols not only helps to make the coding of a program more maintainable and easier to change as needs arise, it also can be used to document the relationships between the various elements of the program.

Figures **8** and **9** show a snippet of code that constructs a dataset name. The name is built from an internal field (TYPENAME), the address space's jobname, and some additional text.

```
 ************************************************************
 * Construct a dataset name (the bad way)                   *
 ************************************************************

 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)))
 *
          ...
          LTORG ,
                =CL44' '
                =C'.GENERAL.PROCESS.LOG'
```

(*Patience. The #TEST macro is discussed later in this paper.)

8 Code with too many magic numbers

In figure **8**, constants are used to represent the lengths of dataset names (44), jobnames (8), and the TYPENAME field (5). Also, hardcoded offsets have been used to refer to fields in the system control blocks, PSA, TCB, and TIOT.

There are two problems with this. The first is, what if you decided to change something in the program? Such as the length of the TYPENAME field, for example? If you did, then in just this little snippet alone, you would have to change the number 5 in three places.

The second problem is that the person reading the program has less information to help in his understanding of the code. Raw numbers are rather uninformative objects. They tend to be devoid of connotations. In the "**`MVC LOGNAME(5),TYPENAME`**" statement, can you tell whether I'm moving all of the TYPENAME field? Or just part of it? In the "L-L-MVC" sequence, can you tell what PSA, TCB, and TIOT fields are being referenced? Can you even tell that the control blocks that are being referenced are the PSA, TCB and TIOT?

```
*************************************************************
* Declare various magic numbers and data areas              *
*************************************************************
DSNAMEL  EQU   44                  LENGTH OF DATASET NAMES
         IHAPSA LIST=YES
         IKJTCB LIST=YES
TIOT     DSECT ,
         IEFTIOT1 ,

*************************************************************
* Construct a dataset name (a better way)                   *
*************************************************************

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'TIOCNJOB,R1),TIOCNJOB APPEND THE JOBNAME
         DROP  R15                  DONE W/TIOT HEADER

         LA    R1,1+L'TIOCNJOB(,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'TIOCNJOB+20,LE,L'LOGNAME)
+        DC    0Y(X'7FFF'-(L'LOGNAME-(L'TYPENAME+1+L'TIOCNJOB+20)))
*        ...
         LTORG ,
               =CL(L'LOGNAME)' '
               =C'.GENERAL.PROCESS.LOG'
```

9 Code with symbols replacing most magic numbers

In figure **9**, as many of the "magic numbers" as possible have been replaced by references to symbols. This makes the program both more self-adjusting, and easier to understand. In particular, the following changes have been made:

- **!** In MVS[1], 44 is a "magic number". It is the maximum length that a dataset name can be. There's no way around it, if you're going to create a dsname buffer, then you just have to use the value "44". But if you must use a magic number, then use it only once. Use it to create an equate. Then use that equate everywhere else that the value is needed. Thus, should you ever have a reason to change 44 to something else, you only have to change it once. That is what is accomplished by the statement, **`DSNAMEL EQU 44`**.

[1] "MVS" is still the name of the operating system at the heart of the bundle of products known as OS/390 and z/OS.

- So the definition ("CL44") of the length of the LOGNAME field has been replaced by a reference to the DSNAMEL equate ("CL(DSNAMEL)").

- Subsequent references to dsname lengths have been replaced by references to the length attribute of the LOGNAME field (instead of to DSNAMEL), since that is the specific field with which this code is dealing.

- References to the length of a job name have been changed from a hardcoded "8" to references to the length attribute of the system control block field from what the job name is being obtained (the TIOCNJOB field of the Task I/O Table).

- References to the length of the TYPENAME field has been changed from a hardcoded "5" to references to the field's length attribute. Thus, if I should want to change the length of the field, I do not also have to change all the instructions and other data that have dependencies upon that length. The assembler will automagically do that for me.

- The references to fields in the PSA, TCB, and TIOT have been changed from hardcoded offsets to field names, not because these offsets will ever change, but to document that these are the control blocks and the fields that are being referenced.

- But in the statement "**`MVC 1(20,R1),=C'.GENERAL.PROCESS.LOG'`**", the length value of "20" has not been changed. This is because the only way to reference the length attribute of the literal would involve repeating the literal within the length field (figure **10**). This is both an ugly and error prone alternative. It's too easy to mistype one or the other of the two instances of the literal.

```
          MVC   1(L'=C'.GENERAL.PROCES.LOG',R1),=C'.GENERAL.PROCESS.LOG'
```

10 Oops! (Not to mention, uuuggggllyy!)

What's a Y-CON?

"**`DC Y(expression)`**" used to be a 2-byte wide address constant. (Yeah, once upon a time mainframes really were that small!). These days, it's little more than a glorified H-CON with one big advantage: Its operand can be an expression.

Documenting Implied Registers

I use Y-CONs heavily in my code, not to carry actually data, but to help document things that aren't otherwise obvious. For example, one time I was reviewing a piece of code wondering why R2 was getting klotzed. I scanned the code looking for possibilities. I checked the cross references for "R2". I used my editor to scan for instances of "R2", I even rescanned for instances of just plain "2", I must have been tired, because it took me somewhere around an hour or so to finally notice a little teeny-weeny TRT hiding amongst the trash. [sigh]

Afterwards, I was moaning about how the assembler should have a way to cross reference implied registers, when it occurred to me that there was no reason at all why I couldn't just do it myself: Y-CONs! From then on, I always followed my TRTs with a couple of zero-length Y-CONs referencing the implied registers (figure **11**). This not only made the usage obvious to the reader, it also documented the references in the assembler's symbol cross reference. ☺

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

Or if you prefer ...

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

11 Using Y-CONs to document a TRT

This usage of Y-CONs, of course, extends to a lot of other instructions. Figure **12** shows a bunch.

```
MR    R0,R3                          MVCSK HERE,THERE
DC    0Y(R1)                         DC    0Y(R0)        LENGTH
                                     DC    0Y(R1)        SOURCE KEY
SRDA  R14,0
DC    0Y(R15)                        SCKPF ,
                                     DC    0Y(R0)
EDMK  whatever
DC    0Y(R1)    1ST DIGIT            CSP   R4,R7
                                     DC    0Y(R5)
TRT   whatever
DC    0Y(R1,R2) HIT PTR/HIT CODE     STFL  0
                                     DC    0Y(L'FLCFACL)  PSA+X'C8'
BXLE  R2,R4,whatever
DC    0Y(R5)    COMPARAND            STSI  SYSIB
                                     DC    0Y(R0)        FUNCTION CODE
IPK   ,                              DC    0Y(R1)        SUBCODE
DC    0Y(R2)
```

12 Using Y-CONs to document implied registers

Documenting Other Unexpected References

But there's no reason to limit Y-CON usage just to implied registers. Figure **13** shows Y-CONs being used to draw attention to "strange conditions".

```
        BXLE  R2,R5,whatever
        DC    0Y(R5)         *BOTH* INCREMENT AND COMPARAND
```

Using the same register both as the length and the comparand in a BXLE is a strange thing to do!

```
        STFL  0
        DC    0Y(L'FLCFACL)  IMPLIED TARGET FIELD (PSA+X'C8')
```

I don't know, maybe it's just me. But I think a machine instruction that has an absolute reference to a specific location in storage is a very strange animal!

13 Using Y-CONs to document strange conditions

Y-CONs can also be used to document implied references to fields (not just registers). The LM and STM instructions, for instance, often refer to multiple registers and to multiple fields of data. The problem is, for any register that is neither

the first nor the last in the range, and for any field that is not the first in the target, the references are hidden from view. They are not reported by the assembler, and they can be very easily overlooked by anyone perusing the code.

```
          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

          STM   R15,R2,SDELWORK     SAVE                          03/00 S20
          DC    0Y(R15,L'SDELTB@L)  @'SYMDEL CONTROL TABLE      03/00 S20
          DC    0Y(R0,SDELSLOT)     TABLE'S SLOT SIZE           03/00 S20
          DC    0Y(R1,L'SDELTBLZ)   Z'TABLE (USED PART)         03/00 S20
          DC    0Y(R2,L'SRCHSDEL)   @'SEARCH CONTROL ROUTINE    03/00 S20
          #TEST SIZE=(L'SDELWORK,GE,4*4) FIT CHECK              03/00 S20
+         DC    0Y(X'7FFF'-(L'SDELWORK-(4*4)))
```

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

14 Using Y-CONs to document the targets of a store multiple (STM)

This problem can be solved by using a **`DC 0Y(...)`** statement to actually reference both the registers and the fields (all of them) affected by the instruction. See figure **14** for a couple of examples, and refer back to figure **6** for another.

Sometimes, I'll used Y-CONs to document subroutine inputs and outputs. Figure **15** shows an example of this.

```
          L     R1,DBCDOPER         Points: 1st operand        11/98 XLQ
          BAS   R14,ARPA64          Decode tgt_addr            06/02 Z12
          B     0(,R15)             +0 ERROR                   11/98 XLQ
******** FALL  THRU                +8 Normal delimiter        11/98 XLQ
          DC    0Y(R0)              PARSED OBJECT'S TYPE       06/02 Z12
          DC    0Y(R1)              L'RESIDUE OF PARSED OBJECT 06/02 Z12
```

15 Using Y-CONs to document subroutine inputs and/or outputs

Y-CONs also are good for xref'ing implied references to flag bits (figure **16**).

```
FLAGS     DC    B'00000000'
REMHUH    EQU   B'10000000'
REMWHAT   EQU   B'01000000'
DOTHIS    EQU   B'00100000'
DOTHAT    EQU   B'00010000'
          ---
          MVI   FLAGS,RMEMHUH               SET THIS
          DC    0Y(REMWHAT,DOTHIS,DOTHAT)   XREF THE ZERO'D FLAGS
```

16 Using Y-CONs to document implied references

Documenting Funky Linkage Conventions

Dontcha just love IBM? They go to all the trouble to define register conventions for subroutines: Branch on R15, link on R14, R13 points to a save area, etc. But do they follow their own rules? Well, sometimes. Sometimes not.

How many of you know, off the top of your head, which registers are preserved and which registers are trashed by the SETLOCK service? I don't. Y-CONs can save a trip to the books. Check out figure **17**.

```
          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
```

17 Using Y-CONs to document funky register requirements (SETLOCK)

How many of you know, of the top of your head, which registers are preserved and which registers are trashed by the Cross Memory POST service? I do. (A lot! See figure **18**.)

```
          POST  (R2),               ECB ADDRESS                 01/94 X22*
                (R3),               POST CODE                   04/93 X22*
                ASCB=(R5),          TARGET A.S.'S ASCB          04/93 X22*
                ERRET=CVTBRET,      NULL ERROR EXIT ROUTINE     04/93 X22*
                ECBKEY=0,           TARGET ECB'S KEY IS ANY     04/93 X22*
                MEMREL=NO,          ERRET RUNS IN MASTER A.S.   04/93 X22*
                LINKAGE=BRANCH      BRANCH ENTRY                04/93 X22
+         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,CVT0PT01-CVTMAP(0,15)
+         BALR  14,15                  . BRANCH ENTRY
          DC    0Y(R0,R1,R2,R3,R4,R5,R6,R7) ONLY R9 SAVED. ALL 06/02 Z12
          DC    0Y(R8,R10,R11,R12,R13,R14,R15) OTHERS TRASHED! 06/02 Z12
```

18 Using Y-CONs to document funky register requirements (cross memory POST)

Clarity of Code

Code should be written in a consistent, straightforward way, and obvious way. Here are a couple of suggestions.

Register Usage

The "volatile" registers are R14, R15, R0, and R1. They're volatile, not because of anything intrinsic in their hardware, but because of the programming conventions regarding their rolls in calling and returning from subroutines.

Volatile registers can be used freely for short-term work data. Even if you have a large chunk of code that does not issue SVCs or call subroutines, please do not use these registers to carry long-term data. Who knows, someday someone might want to throw a service macro into your code, and he just might not notice that R1 was set to a critical value two pages ago that's going to become very important a page or three later on. (Oops!)

In fact, this kind of problem is not limited to the volatile registers. Except for program bases and data area pointers, registers simply should not be used to contain long-term values. Having to search back through 200 lines of code to find the setting of a register does not contribute to clarity.

Work registers should contain values that are only of local or current use.

If a work register contains a value that is needed in separated units of logic, then that value should be saved and reloaded from a data field.

Using USINGs

USINGs are great devices. Not only do they make symbol usage possible, when used precisely, they also serve as a valuable documentation aid of what a register is being used for and when it is being used.

USINGs should be declared for all registers that contain long term pointers. If a register contains a long term value for which a USING is not appropriate, then consideration should be given to keeping that value in a data field rather than in a register.

Whenever any register contains a pointer to a data area, a USING should be issued to declare that relationship.

Whenever a register no longer contains a valid pointer, that USING should be DROP'd.

When a piece of code uses a particular register to point to a particular data area only intermittently, please do not just issue a USING at the start of the code and a DROP at the end. Instead, issue a USING whenever the register actually does contain a pointer and a DROP when it does not.

Sometimes you might have a routine wherein a register may or may not contain a pointer, and for those portions of the routine where it matters, the code has to perform a test. In this case, the register will be known to contain the pointer only for certain sections of the routine and will be considered to be "reserved" otherwise. To document this, do not just issue a USING at the start of the code and a DROP at the end. Instead:

- Create a null dsect named RESERVED to be used for identifying reserved registers.
- Issue a named USING to assign the RESERVED dsect to the register at the start of the routine. Example: **`RES_R8 USING RESERVED,R8`**. This USING documents that the register is reserved throughout the routine.
- DROP the named USING at the end of the routine. Example: **`DROP RES_R8`**.
- Then use unnamed USINGs and DROPs to document exactly when the pointer is known to be valid and when it is not. (These unnamed USING/DROPs and the named USING/DROP will not interfere with each other.)

Figure **19** shows an example of this technique.

```
RESERVED DSECT ,                    FOR DECLARING RSERVED REGS 10/89 X21
DBCSRSVC CSECT ,                    RESUME CODE SECTION        10/89 X21
         ...
         ...
         SLR   R8,R8                CLEAR THE SDWA PTR REG     10/89 X21
RES_R8   USING RESERVED,R8          NOTE THAT IT'S RESERVED    02/02 Z12
         CH    R0,=Y(12)            IS AN SDWA AVAILABLE?      10/89 X21
         BE    NOSDWA1              NO, PROCEED                08/90 X21
         LR    R8,R1                YES, COPY ITS ADDRESS      10/89 X21
         USING SDWA,R8              DECLARE ITS BASE           10/89 X21
         L     R1,SDWAABCC          GET ABEND CODE             01/98 X34
         ...
         DROP  R8                   SUSPEND SDWA BASE          08/90 X21
NOSDWA1  DS    0H                                              01/98 X34
         ...
         LTR   R8,R8                SDWA EXIST?                08/90 X21
         BZ    ABDORTRY             NO, SKIP                   07/97 X34
         USING SDWA,R8              YES, DCL SDWA BASE         07/90 X21
         OI    SDWAACF2,SDWARCRD    REQUEST LOGREC RECORDING   08/90 X21
         OI    SDWACMPF,SDWAREQ     REQUEST ABEND DUMP         08/90 X21
         DROP  R8                   KILL SDWA BASE             07/90 X21
NOSDWA1  DS    0H                                              07/97 X34
         ...
         LTR   R1,R8                IS THERE AN SDWA?          10/89 X21
         BZR   R14                  NO, JUST RETURN            08/90 X21
         SETRP RC=0,                YES, SET RETURN CODE       10/89 X21*
               DUMP=YES,            PRODUCE A DUMP             10/89 X21*
               RECORD=YES,          RECORD TO LOGREC           10/89 X21*
               WKAREA=(R1)          R1 POINTS TO THE SDWA      10/89 X21
         BR    R14                  RETURN TO RTM TO PERCOLATE 08/90 X21
         DROP  RES_R8               DONE W/R8                  02/02 Z12
```

19 Declaring a register as being "reserved".

Complexity of Comprehension

Do not add unnecessarily to the amount of information the reader needs for understanding a piece of code. Be considerate of the reader! Figures **20** and **21** show an example involving setting and clearing flags. One programmer I know insisted upon using the XI instruction to clear flags that he knew to be on and to set flags he knew to be off. What's wrong with that? Well, suppose he's wrong ...

```
         ...
         OI    THKFLAGS,THKFALES    assume: Do not delete      09/99 XLQ
         ...
         ...   59 nine lines of code
         ...    and 8 labels
         ...      later.
         ...
         XI    THKFLAGS,THKFALES    Remember to delete ALET.   09/99 XLQ
         ...
```

What's the intent here? Turn the bit on? Turn it off? Flip it? Who knows? Well, I guess this programmer knows, and he says that his intent is to turn it off. But it's a pain in the butt to confirm his assertion.

20 On or off, which is it?

```
         ...
         OI    THKFLAGS,THKFALES   assume: Do not delete      09/99 XLQ
         ...
         ...   59 nine lines of code
         ...    and 8 labels
         ...     later.
         ...
         NI    THKFLAGS,255-THKFALES Remember to delete ALET. 07/02 DBC
         ...
```

Here, there's no ambiguity. The programmer intends to turn the flag off, and I believe him.

21 Turn it off for sure!

Except for loops, the progression of execution should be generally forward in the code. Hopscotching back and forth complicates understanding. Figure **22** shows a simplified example. Rather than putting an error handler near the end of a subroutine, the programmer has placed it at the point of first usage. Subsequent users of that handler have to branch back to get to it. In large routines, this technique, used many times, can lead to convoluted and confusing code. A better choice would be to gather all error handlers towards the bottom of the routine.

```
*        ...
         GETMAIN RC,whatever       GET A WORK AREA
         LTR   R15,R15             OK?
         BZ    GOTWA               YES, PROCEED

GETMFAIL TPUT  'GETMAIN FAILURE'   SEND MSG
         B     SUBRET0             GO MAKE ERROR RETURN.
GOTWA    DS    0H

*        ...
         GETMAIN RC,whatever       GET AN INPUT BUFFER
         LTR   R15,R15             AOK?
         BNZ   GETMFAIL            NO, GO HANDLE
*        ...
SUBRET0  ...
```

22 Back and forth, back and forth

Figure **23** shows a rather strange convolution arising from the programmer's desire to save four bytes of code. It certainly confused me the first time a saw it. And it still causes me to scurry back and forth in the listing whenever I have to troubleshoot code that uses this technique.

```
1st)             ...
2nd)  XMSMODE    BAS   R15,SHIELD1          GO CREATE STACK SHIELD 07/98 XLQ
4th)             ...   500
                 ...    lines
                 ...     of code.
                 ...
5th)             PR    ,                    DELETE STACK SHIELD    07/98 XLQ
3rd)  SHIELD1    BAKR  0,R15                CREATE STACK SHIELD    07/98 XLQ
6th)             ...
```

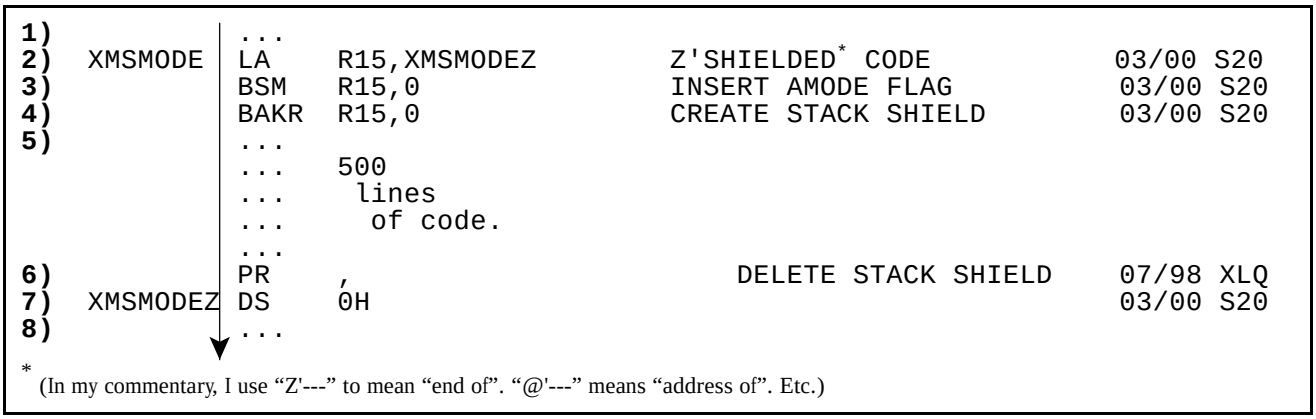
Clever but ...

Bytes saved? Four.

Support programmers lost? At least one.

23 Down, then up, then down, then ... huh?

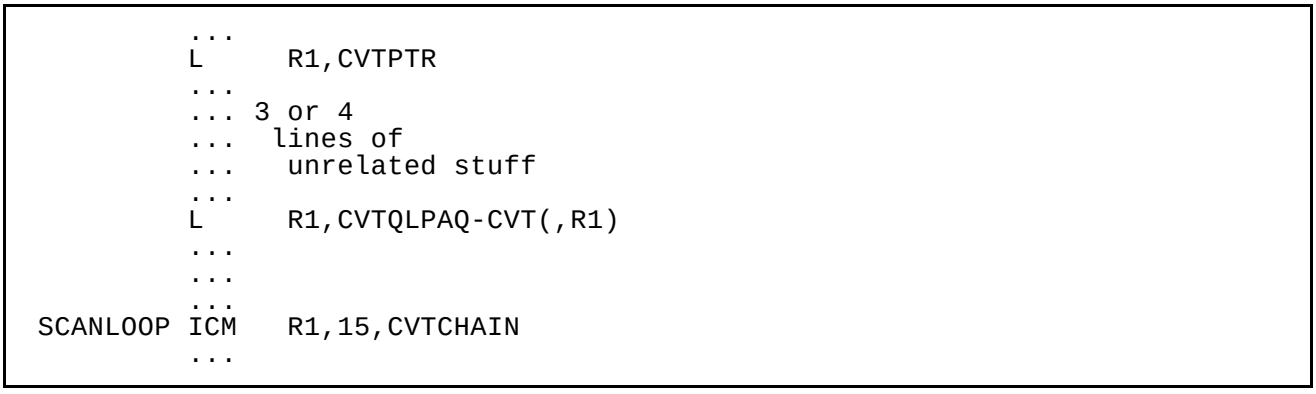
Figure **24** shows a better way to accomplish the same result. Here, execution proceeds in a straight and forward manner. The scurrying factor has been reduced considerably.



24 Linear code

Performance Coding

Is coding for performance really that good of an idea? It depends. Unfortunately, on current machines, performance and clarity have come into conflict like never before. With pipelining, instructions can be processed much faster than registers can be loaded or copied. Consequently, a delay needs to occur between the loading of a register and the use of that register. So in order to achieve the most efficient use of machine cycles, useful but unrelated instructions need to be inserted, if possible, so as to space out the loading of and the usage of a register. (See figure **25**. [sigh]) And I suppose, the degree to which this matters differs from one machine to another. [heavy sigh]



25 Efficient? Well, I suppose ...

So the question arises, what's more important, Clarity or efficiency? Again, it depends. The first question to ask: Is your code compute bound or I/O bound. If compute bound, then efficiency probably does matter more than clarity. If I/O bound, then it probably does not.

The next question to ask: How much real world time is a particular code change going to save? Are you pouring out sweat and working hours and hours to squeeze performance out of initialization code? Is your successor pouring out sweat and working hours and hours to understand that code? Congratulations, you've just saved about a microsecond per year of processing time.

Other questions: How long does it take your program to run? How often is it going to be run? Are there real world dollars involved? If your program is going to be executed once a day, and it only takes 10 seconds to execute, what's the point in making it run in 2 seconds? (Real dollars? Well, then maybe.)

If you're writing code that's in critical loops in a lengthy compute bound process, or if you're writing macros that are going to be used to an unknown degree in unforseen programs, then sure, coding for efficiency is probably the overriding aesthetic. Otherwise, the development and maintenance costs probably aren't worth it. You have to use good judgement.

Also, keep in mind that the paradigm of efficiency has changed considerably over the years, and probably will change again. For example: Once upon a time, efficient dataset I/O was important. But then one vendor wrote chained channel programs for tape access that were so efficient, that they monopolized the channels and locked out all other devices. I suppose they were very proud of how "good" their channel programs were until their customers forced them to throttle back. ... And now, there's caching. That goes a long way towards reducing the importance of efficient user program coding.

Remember, technology changes. How long will it be before pipelining techniques change or are replaced by something new and not yet thought of by you and me?

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 the program, the logic of the program, the intent of the code, or the current environment. Sanity Checks are a very important part of documentation. When troubleshooting, the presence of sanity checks can save a lot of time and effort by guiding the programmer away from irrelevant inquiries.

Assembly Time Sanity Checks

At Assembly time, a sanity check is any usage that leads to a syntax error, should a required condition not be met. A lot of sanity checks can be accomplished simply by proper usage of symbols. For example, in figure **26**, there is a requirement that the length of MYNAME be less than 9. If that is not true, then a syntax error will occur on the 2nd MVC:[1]

```
MYNAME   DC    C'XDC'
MYNAMEL  EQU   L'MYNAME
         ...
         MVC   DBCDWORK(MYNAMEL),MYNAME BUILD THE DEFAULT -   09/99 S11
         MVC   DBCDWORK+MYNAMEL(8-MYNAMEL),=CL8'SLIST' -      09/99 S11
*                                SLIST-LIBRARY DDNAME        05/92 X22
```

26 Typical assembler capable consistency check

But there are many situations that arise in which the assembler has no direct ability to detect a problem. For example, if you're constructing a message into a buffer, there is a need to insure that the message does not overflow the buffer. If the message is built from a series of constants and fixed length elements, then there is no easy way for the assembler to directly check for overflow.

However, there is an easy way create an assembler check that is sensitive to the overflow. Simply use the length of the message and the length of the buffer to concoct a constant whose value will be syntactically invalid if (and only if) the message is too long. If you make it a zero-length constant, then it can even be placed inline at the point in the code where the tested condition actually matters. Figure **27** shows what I mean. The Y-CON accepts a maximum value of 32767.

[1] Actually, there's a bug here. I know what it is. Do you?

The arithmetic used in the example produces a legal value if, and only if, the length of LEQMSG is less than or equal to the length of DBCMSG.

```
          MVC   DBCMSG(L'DBCMSG),BLANKS CLEAR MSG BUFFER       06/98 X34
LEQM      USING LEQMSG,DBCMSG       DCL MSG FIELDS OVERLAY     06/98 X34
          DC    0Y(X'7FFF'-(L'DBCMSG-L'LEQMSG)) DOES LEQMSG    06/98 X34
*                                   FIT INTO DBCMSG?           06/98 X34
```

27 Detecting buffer overflow

I have written a macro to perform this service. It is named #TEST, and it can be downloaded from "**www.colesoft.com/utilities.html**". Just click on the "*macros.zip*" oval. An example of its usage is shown in figure **28**.

```
          TM    DBCWMODU+L'CDNAME,255-C' ' IS NAME TOO LONG    06/98 X34
*                                   FOR "LOAD"?                06/98 X34
          #TEST SIZE=(L'DBCWMODU,GT,L'CDNAME) (DEPENDENCY CHK) 06/98 X34
          DC    0Y(X'8000'-(L'DBCWMODU-(L'CDNAME)))
```

28 Insuring field length dependencies

#TEST allows you to specify any comparison between two assembler values or expressions. If generates the one or two Y-CONs necessary to prove the truth of that comparison. If the relation is false, then the Y-CON's value limit will be exceeded, and the assembler will complain.

Sanity Check Example: Work Buffer Overflow

Many of my subroutines are provided with a short, fixed length work area for their private use. So each such routine has a small dsect map that defines that routine's use of its work area. (There are hundreds of these.) It is important that a routine's use of the work area not exceed its length. Figure **29** shows #TEST being used to verify and document a proper fit.

```
 ************************************************************ 10/99 S11
 * Local RSTKWORK fields                                    * 12/00 S21
 ************************************************************ 12/00 S21
 ATWNWORK DSECT ,                                            12/00 S21
 ATWN     @ADDRESS ALET,ASID                                 06/02 Z12
+ATWNALET DC    F'0'                ALET                     04/02 Z12
+ATWNAS_F DC    F'0'                ASID (FWORD)             04/02 Z12
+ATWNASID EQU   *-2,2               ASID (HWORD)             04/02 Z12
+ATWN_ALAS EQU  ATWNALET,*-ATWNALET                          04/02 Z12
 ATWNIRB# DS    H                   IRB'S #                  12/00 S21
 ATWNFLGS DS    B                   LOCAL FLAGS              12/00 S21

          #TEST SIZE=(*-ATWNASID,LE,L'RSTKWORK) FIT CHECK    12/00 S21
          DC    0Y(X'7FFF'-(L'RSTKWORK-(*-ATWNASID)))
 DBCMISC  CSECT ,                   RESUME CODE SECTION      12/00 S21
          ...
          ...
          ...
          USING RSTK,RSTKREG          DCL GENERIC RSTK BASE  10/99 S11
          USING ATWNRSTK,RSTKWORK     DCL LOCAL WORK FIELDS  12/00 S21
          ...
```

29 Detecting a work buffer overflow

Sanity Check Example: Table Update Reminder

My product has a table in it to which an entry must be added each time a new release or version is created. (See figure **30**.) I use #TEST to remind me to do this whenever I change the release number variable.

```
 ************************************************************ 12/94 X31
 * The following table is used to determine the release of   * 12/94 X31
 * older XDC's by their load module lengths. (Sorted)        * 06/96 X32
 ************************************************************ 12/94 X31
 OLDXDCS  DS    0F                  ALIGNMENT                  12/94 X31
          DC    X'0000DD68',C'X3.3' X33                        10/97 X34
          DC    X'00010060',C'S1.0' S10                        01/99 XLQ
          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=(&XDCVER#,EQ,12) ALERT WHEN RELEASE       12/01 Z12*
                                      CHANGES FROM z1.2        12/01 Z12
+         DC    0YL2(X'7FFF'-(12)+12,X'7FFF'-(12)+12)
```

30 Version/release dependency

Sanity Check Example: Register Usage Dependency

Throughout my product, the register named TWAREG always points to the local routine's Temporary Work Area. It just so happens that TWAREG equals R9. At one point, I invoke a cross memory POST routine which just so happens to trash every register except R9. What a lucky coincidence! The TWA is where I've stashed the registers that I'll need to restore.

As shown in figure **31**, I can use the #TEST macro to document that coincidence, and to insure that I'll be alerted should I (or a successor) ever change the value of TWAREG. (I'll just have to trust that IBM will never damage POST's API.)

```
          POST  (R2),               ECB ADDRESS                01/94 X22*
          ...
                LINKAGE=BRANCH      BRANCH ENTRY               04/93 X22
+         ...
+         BALR  14,15                  . BRANCH ENTRY                @D1A
          DC    0Y(R0,R1,R2,R3,R4,R5,R6,R7)    XREF TRASHED -  06/02 Z12
          DC    0Y(R8,R10,R11,R12,R13,R14,R15)  REGISTERS.     06/02 Z12
          DROP  ,                   ALL REGS (BUT R9) GONE     04/93 X22
          USING TWAPOST,TWAREG      DCL TWA BASE               04/93 X22
          #TEST SIZE=(TWAREG,EQ,9)  DEFINITION CHECK           04/93 X22
+         DC    0YL2(X'7FFF'-(TWAREG)+9,X'7FFF'-(9)+TWAREG)
```

31 Document a dependency

Sanity Check Example: Control Block Version Dependency

The High Level Assembler has published three different versions of ADATA. Each ADATA record has a header field. For versions 1 and 2, the header fields were identical, but in version 3, they changed.

My product attempts to understand all three versions of ADATA, and so it has dependencies upon what changes occurred in which versions. Figure **32** shows an example of using a #TEST macro to document a piece of code wherein the lengths of the V1 and V2 header fields are expected to be the same.

```
          #TEST SIZE=(V2_ADATA_HEADERL,EQ,V1_ADATA_HEADERL)    04/99 S11*
                                INSURE ADATA V1/V2 HEADERS  04/99 S11*
                                ARE SIMILAR                 04/99 S11
+         DC    0YL2(X'7FFF'-(V2_ADATA_HEADERL)+V1_ADATA_HEADERL,X'7FFF'*
+               -(V1_ADATA_HEADERL)+V2_ADATA_HEADERL)
```

32 Document that the V1/V2 ADATA headers are the same

Sanity Check Example: Chain Field Location Dependency

Figure **33** shows some code that has a logic dependency that the CDE's chain field must be the CDE's first field. The #TEST macro documents that dependency.

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

33 Document logic dependencies

Execution Time Sanity Checks

At execution time, sanity checks can range from being corrective error handlers to abortive program check traps. Corrective error handlers, of course, generally are pretty complex, and certainly must be written for any error condition that has any likelihood of occurring. Abortive traps, on the other hand, are trivially easy to construct, but should only be used for conditions that you truly believe should never occur. (Of course, you may be wrong, in which case you're going to be real glad that you had the trap!)

Typically, an abortive trap is an instruction that will cause an immediate program check in the event that some condition occurs that really really shouldn't occur. The condition is considered either to be impossible or to be so unlikely that the effort of writing a handler is not worth taking. Such traps also can be used to guard against future code changes that might be incompatible with current logic. Figure **34** shows examples of several types of abortive traps.

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)
```

Executing a #DIE macro permits a condition test and causes a stylized 0C1 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    0H
```

34 Examples of commonly used abortive traps

I have written a macro to generate conditional, abortive traps that can have explanatory messages associated with them. The macro is named #DIE, and it generates what I call a "DEAD-trap". Depending upon the operands given, the DEAD-trap can be generated either in-line (with a conditional skip around it) or out-of-line as a literal operand of the generated branching instruction. If you want the DEAD-trap to include a special message, then you can provide that message, in a quoted string, as the macro's last operand; otherwise, a default message will be generated that will be a unique 4-digit number.

#DIE can be downloaded from "**www.colesoft.com/utilities.html**". Just click on the "*macros.zip*" oval. Its operands and usage are fully documented by internal commentary.

Abortive traps are wonderfully useful devices. Here are some examples of the situations in which they can be used.

DEAD-Trap Sanity Check Example: Recursion Underflow Trap

I'll use DEAD-traps to catch illogical conditions early before they have a chance to wreck havoc or to hide themselves behind million-instruction histories.

Stack underflow is an example of an illogical condition. If a program is written correctly, then underflow should NEVER occur. Nonetheless, it is both a safe and comforting thing to make sure. A DEAD-trap comes in handy here. Figure **35** shows an example.

```
 ********************************************************** 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
 ADDRRET0 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    0H                  RECEIVE SKIP AROUND TRAP
```

35 Abort upon recursion stack underflow

DEAD-Trap Sanity Check Example: Documenting Illogical Conditions

When troubleshooting a problem, the presence of DEAD-traps that have not been executed help to guide me away from irrelevant inquiries. That can be of considerable help in the debugging process. (See figure **36**.)

```
 ********************************************************** 08/99 S11
 * Check for GOFF records that contain ADATA. Ignore those   * 08/99 S11
 * that do not.                                              * 08/99 S11
 ********************************************************** 08/99 S11
          ...
          SR    R4,R2               GET L'OBJ DATA (+PADDING   08/99 S11
 *                                  IF ANY) ON THIS CARD.      08/99 S11
          #DIE  NP                  (LOGIC ERROR)              08/99 S11
+         BP    DIE5621Z            SKIP IF OK
+         DC    X'00DEAD',AL1(4),C'5621'                       02/99 XLQ
+DIE5621Z DS    0H                  RECEIVE SKIP AROUND TRAP
```

36 Guarding against an illegal condition that's conceivable but is too unlikely to be worth coding for

DEAD-Trap Sanity Check Example: Enforcing Interface Dependencies

DEAD-traps are great for documenting interface dependencies. Are my inputs what the caller promised? Are a subroutine's outputs what they're supposed to be? Generally, I'll use these kinds of traps at interfaces to subroutines that I might not fully trust and over which I have no control, such as subroutines into other products. I'll also use them for subroutine calls within large complex products. A couple of examples are shown in figures **37** and **38**.

```
 ************************************************************ 04/99 S11
 * Make sure I'm being called with the right TWA (TWAMAP).   * 04/99 S11
 ************************************************************ 04/99 S11
          CLI   TWATYPE,TWATMAP     "TWAMAP" TWA?              04/99 S11
          #DIE  NE                  NO, LOGIC ERROR            04/99 S11
+         BE    DIE5618Z            SKIP IF OK
+         DC    X'00DEAD',AL1(4),C'5618'                       02/99 XLQ
+DIE5618Z DS    0H                  RECEIVE SKIP AROUND TRAP
```

37 Interface check: Verify my caller's environment

```
 ************************************************************
 * Recurse. i.e. resolve the given dsect's base address     *
 * string.                                                  *
 ************************************************************
          BAS   R14,GETADR64        RESOLVE THE ADDRESS STRING 01/02 Z12
          B     DMBERRMA            +0 ERROR; GO HANDLE        01/02 Z12
          #DIE  C,15                +4 SPECIAL DLM; CAN'T HAPN 01/02 Z12
+         BC    15,=C'....7032'*                              10/99 XLQ
 ******** FALL  THRU                +8 NORMAL DLM; AOK         01/02 Z12

*
  Notes:
      ! The "...." represents binary data. Specifically, it represents X'00DEAD',AL1(4).
      ! An out-of-line DEAD-trap is use here because the #DIE macro is being issued from within a subroutine's return vector, and so
        the #DIE macro must generate in-line code that is exactly 4 bytes wide, no more, no less. (An in-line DEAD-trap would generate
        at least 8-bytes.)
```

38 Weed out unexpected returns

DEAD-Trap Sanity Check Example: Future Code Expansion Points

DEAD-traps can be used to document code points for future expansion (and to catch the error of only a partial implementation). For example, my product will read ADATA from either of two sources, a SYSADATA file, or a GOFF type object file. Should a third ADATA source ever be thought of, I'll have to update my product. Figure **39** shows a #DIE documenting the code insertion point. (It also will trap execution if my type flags are, for some reason, not set.)

```
 ************************************************************ 08/99 S11
 * GOFF record processing.                                  * 08/99 S11
 ************************************************************ 08/99 S11
          SPACE 1                                             08/99 S11
          TM    ADATFLGS,ADFGOFF    GOFF PROVIDER?            08/99 S11
          BZ    SMGOFFZ             NO, SKIP                  08/99 S11
          ...
 SMGOFFZ  DS    0H                                            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   0H                                            03/00 S20

 ************************************************************ 08/99 S11
 * Unknown record processing. Development error.            * 08/99 S11
 ************************************************************ 08/99 S11
          #DIE  ,                                             08/99 S11
+         NOP   DIE5622Z            GATE
+         DC    X'00DEAD',AL1(4),C'5622'                      02/99 XLQ
+DIE5622Z DS    0H                  RECEIVE SKIP AROUND TRAP
```

39 Guarding against future record source types

DEAD-Trap Sanity Check Example: Buffer Overflow, Table Overflow, Etc.

DEAD-traps can also be used to guard against the unexpected exceeding of a buffer size or some other limit. Figures **40** and **41** show a couple of examples of this.

```
 ************************************************************ 08/99 S11
 * Now getmain, build, and queue a new cache block.         * 08/99 S11
 ************************************************************ 08/99 S11
 SCGETBLK L     R1,=A(ADCBDSIZ-ADCRECRL) CACHE BLOCK'S MAX    08/99 S11
 *                                DATA CAPACITY              08/99 S11
          CLR   R5,R1              WILL THE ADATA RECORD FIT? 08/99 S11
          #DIE  H,'HUGE ADATA RECORD ENCOUNTERED!' NO, IF THIS 08/99 S11*
                                   HAPPENS SOMEDAY, I'LL NEED 08/99 S12*
                                   TO FIX IT.                 08/99 S12
+         BNH   DIE5627Z           SKIP IF OK
+         DC    X'00DEAD',AL1(35),C'5627 HUGE ADATA RECORD ENCOUNTERED!'
+DIE5627Z DS    0H                RECEIVE SKIP AROUND TRAP
```

40 Execution-time buffer capacity check

```
 ************************************************************* 03/00 S20
 * Assembly change detection - If the current XREF record is * 03/00 S20
 * from a new assembly, then insert an assembly boundary     * 03/00 S20
 * entry. This entry resets the SYMDEL nest level to zero    * 03/00 S20
 * for the new assembly. Also, keep track of an assembly     * 03/00 S20
 * high water mark. (Probably not necessary since the code   * 03/00 S20
 * that creates the assembly numbers does so in ascending    * 03/00 S20
 * order. But it doesn't hurt to be flexible, or paranoid.)  * 03/00 S20
 ************************************************************* 03/00 S20
          CL    R4,ADCRASM#         SAME ASSEMBLY?            03/00 S20
          BE    SDBASMZ             YES, PROCEED              03/00 S20
          CL    R2,SDELTBLZ         TABLE OVERFLOW?           03/00 S20
          #DIE  NL                  YES, LOGIC ERROR          03/00 S20
+         BL    DIE5632Z            SKIP IF OK
+         DC    X'00DEAD',AL1(4),C'5632'                      02/99 XLQ
+DIE5632Z DS    0H                  RECEIVE SKIP AROUND TRAP
```

41 Guarding against a conceivable but unexpected table overflow

DEAD-Trap Sanity Check Example: Documenting/Verifying Complex Code

DEAD-traps are also good for verifying and documenting complex code or uncertain logic. They can be used to prove and document subtle or indirect relationships and inferences. For uncertain code, the presence of double checking logic and DEAD-traps will, over time, either prove you wrong or create confidence that you were right.

```
 ************************************************************* 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
          BZ    SCFRDUEZ            NO, SKIP THE FREEMAIN     08/99 S11
          #DIE  M                   (LOGIC ERROR)             08/99 S11
+         BNM   DIE5626Z            SKIP IF OK
+         DC    X'00DEAD',AL1(4),C'5626'                      02/99 XLQ
+DIE5626Z DS    0H                  RECEIVE SKIP AROUND TRAP
```

42 Illogical condition: Absent corruption, the result cannot be negative

```
          ...
 ************************************************************       X20
 * Plus/Minus Processing - At this point, OPERATOR is on and * 12/01 Z12
 * INDIRECT is off. Therefore, the current character must be * 10/93 X22
 * a plus (+) or minus (-). This means that an offset        * 10/93 X22
 * expression follows. Call a subroutine to process it.      * 10/93 X22
 ************************************************************       X20
          LA    R0,XPPLUMIN         PRIORITY OF +/-            10/93 X22
          L     R15,=A(OPERATRT)    OPERATOR INDEX TRT         10/93 X22
          SLR   R2,R2               CLEAR FOR TRT              10/93 X22
          TRT   0(1,R1),0(R15)      GET INDEX OF +/-           10/93 X22
          DC    0Y(R1,R2)           XREF IMPLIED REGISTERS     12/01 Z12
          #DIE  Z                   LOGIC ERROR IF NO HIT      10/93 X22
+         BNZ   DIE7022Z            SKIP IF OK
+         DC    X'00DEAD',AL1(4),C'7022'                       02/99 XLQ
+DIE7022Z DS    0H                  RECEIVE SKIP AROUND TRAP
```

43 Documenting an illogical condition

Summing Up

Documentation. Symbol usage. Sanity checks. Clarity of thought and style. It may take extra development time, but if you can afford the time, the long-term costs will be reduced, and the life and usefulness of the product will be extended.

Paranoia! In programming, that's a good thing.