cole software® 736 Fox Hollow Road Afton, VA 22920 Phone: 540-456-4595 www.colesoft.com

Using USINGs
What the Huh????

Share, August 2005
Session 8161
(Thursday, 11:00AM)

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

(Generated August 11, 2005)

Using USINGs - What the Huh????

Contents

Figures

Using USINGs - What the Huh????

The majority of machine instructions that reference storage do so through something called “base/displacement addressing”. This means that in order for most instructions to refer to a particular location:

- A register has to already contain an address that is “near” that location,
- And then you have to form the machine instruction that uses that register and that says how far the desired location is from the place pointed to by that register.

The process of doing this is called “resolving the address”. It's rather tedious.

```
                                    76 ****************************************
                                    77 *                                      *
                                    78 * Sample Program -- Basic machine      *
                                    79 * instructions. No symbols at all.     *
                                    80 *                                      *
                                    81 ****************************************

000000             00000 000B0      83 CODE     CSECT ,
000000 47F0 F028          00028     84          B     40(,R15)
000004 23                           85          DC    AL1(35)
000005 C485949640979996             86          DC    C'Demo program for '
000016 E4A289958740E4E2             87          DC    C'Using USINGs class'

000028 90EC D00C          0000C     89          STM   R14,R12,12(R13)
00002C 41C0 F030          00030     90          LA    R12,48(,R15)
000030 4110 C034          00034     92          LA    R1,52(,R12)
000034 50D0 1004          00004     93          ST    R13,4(,R1)
000038 5010 D008          00008     94          ST    R1,8(,R13)
00003C 18D1                         95          LR    R13,R1

00003E 58F0 C028          00028     97          L     R15,40(,R12)
000042 5AF0 C02C          0002C     98          A     R15,44(,R12)
000046 50F0 C030          00030     99          ST    R15,48(,R12)

00004A 58D0 D004          00004    101          L     R13,4(,R13)
00004E 98EC D00C          0000C    102          LM    R14,R12,12(R13)
000052 17FF                        103          XR    R15,R15
000054 07FE                        104          BR    R14
000056 0000
000058 075BCD15                    106          DC    F'123456789'
00005C 3ADE68B1                    107          DC    F'987654321'
000060                             108          DS    F
000064                             109          DS    XL72
                                   111          END   ,
```

1 Sample Program -- Pure machine instructions: No symbols at all

The sample program shown in figure 1 contains several machine instructions that use base/displacement addressing. Here are a couple of examples:

- **`STM R14,R12,12(R13)`**
  This says, “Store a bunch of registers starting at the location that's 12 bytes past the location pointed to by R13.” The “12(R13)” part of the instruction is referred to as an S-CON. It is converted to the “**`D00C`**” part of the machine instruction. “12(R13)” defines the **base** location (the address pointed to by R13) and the

displacement (the value 12) to be added to that location. In this case, that works out to a place that is outside of the program. (It is a location that has been provided by the program's caller.)

- **`A R15,44(,R12)`**
  This says, "Add, to the contents of R15, a value located at 44 bytes past the location pointed to by R12." The S-CON part of this instruction is "44(,R12)" ("**`C02C`**" in the machine instruction). The **base** is the location pointed to by R12, and the **displacement** is 44. In this program, that works out to be the **`DC F'987654321'`** statement located at +00005C from the start of the code.

- In fact, almost all of the instructions in this example use base/displacement addressing. These instructions are shown in figure 2. The **boldface** text shows the S-CON (base/displacement) portions of the instructions, both in the human-readable format and the object code format.

Machine Instructions Human Readable Format		**Machine Instructions Object Code Format**	
B	**40(,R15)**	47F0	**F028**
STM	R14,R12,**12(R13)**	90EC	**D00C**
LA	R12,**48(,R15)**	41C0	**F030**
LA	R1,**52(,R12)**	4110	**C034**
ST	R13,**4(,R1)**	50D0	**1004**
ST	R1,**8(,R13)**	5010	**D008**
L	R15,**40(,R12)**	58F0	**C028**
A	R15,**44(,R12)**	5AF0	**C02C**
ST	R15,**48(,R12)**	50F0	**C030**
L	R13,**4(,R13)**	58D0	**D004**
LM	R14,R12,**12(R13)**	98EC	**D00C**
MVC	**12(**38**,R5),59(R2)**	D225	**500C 203B** (not in the sample program)

Base/Displacements and their resulting S-CONs are shown in **boldface** text.

2 Machine instructions with Base/Displacements

S-CONs

In object code format, an S-CON is a 2-byte value that may be found in either the 2nd or 3rd halfword of a machine instruction. Its first digit identifies the register being used as a base, the remaining three digits identify the displacement value to be added to the contents of the register to form the address to be referenced.

What's lacking in figure 1, of course, are symbols. Without symbols, the programmer has to create hardcoded S-CONs in order to address locations in storage. Figure 3 shows the same program, but now with symbols added (but not yet being used).

Using USINGs - What the Huh????

```
                                    76 ****************************************
                                    77 *                                      *
                                    78 * Sample Program -- With Program       *
                                    79 * Symbols Added, But Not Being Used.   *
                                    80 *                                      *
                                    81 ****************************************

000000             00000 000B0      83 CODE     CSECT ,
000000 47F0 F028       00028      84          B     40(,R15)
000004 23                         85          DC    AL1(ICATCHRZ-ICATCHR)
000005 C485949640979996           86 ICATCHR  DC    C'Demo program for '
000016 E4A289958740E4E2           87          DC    C'Using USINGs class'
000028                            88 ICATCHRZ DS    0H

000028 90EC D00C       0000C      90          STM   R14,R12,12(R13)
00002C 41C0 F030       00030      91          LA    R12,48(,R15)
000030                            92 BASE     DS    0H

000030 4110 C034       00034      94          LA    R1,52(,R12)
000034 50D0 1004       00004      95          ST    R13,4(,R1)
000038 5010 D008       00008      96          ST    R1,8(,R13)
00003C 18D1                       97          LR    R13,R1

00003E 58F0 C028       00028      99          L     R15,40(,R12)
000042 5AF0 C02C       0002C     100          A     R15,44(,R12)
000046 50F0 C030       00030     101          ST    R15,48(,R12)

00004A 58D0 D004       00004     103          L     R13,4(,R13)
00004E 98EC D00C       0000C     104          LM    R14,R12,12(R13)
000052 17FF                      105          XR    R15,R15
000054 07FE                      106          BR    R14
000056 0000
000058 075BCD15                  108 FIRST#  DC    F'123456789'
00005C 3ADE68B1                  109 SECOND# DC    F'987654321'
000060                           110 ANSWER  DS    F
000064                           111 SAVEAREA DS   XL72
                                 113          END   ,
```

3 Sample Program -- With Program Symbols Added (but Not Being Used)

In figure 3, the instruction "**`A R15,44(,R12)`**" references the statement "**`SECOND# DC F'987654321'`**" located at offset +X'00005C' (92 decimal). The S-CON that accomplishes this reference is 44(,R12).

It's worth while to examine exactly how this S-CON was arrived at. Why R12? Why 44?

- Why R12? The machine instruction "**`LA R12,48(,R15)`**" (at offset +00002C) caused R12 to point to the start of the program plus 48 (offset +000030 in hex, where I've inserted the **`BASE DS 0H`** statement).
- Why 44?
  - 44-decimal is 00002C-hex.
  - R12 points to offset +000030.
  - The **`DC F'987654321'`** statement is located at +00005C, which is equal to +000030+00002C.

Using USINGs - What the Huh????

Using Symbols

Of course, it's not feasible to write programs using nothing but hardcoded S-CONs. Calculating them by hand is somewhat arduous. Recalculating them whenever the program changes is no fun either. But the assembler can do this kind of drudge work quite well, so let's shift this burden from us to it.

```
          Offset     Symbol
R15-->*   +000000   CODE
           ...
          +000005   ICATCHR
           ...
          +000028   ICATCHRZ
           ...
          +00002C             LA     R12,48(,R15)
           ...
R12-->**  +000030   BASE
           ...
          +00003E             L      R15,40(,R12)
           ...
          +00004E             LM     R14,R12,12(R13)
           ...
          +000058   FIRST#
          +00005C   SECOND#
          +000060   ANSWER
          +000064   SAVEAREA
```

* When the program begins, R15 points to its start. This remains true until the **L R15,40(,R12)** statement is reached at offset +00003E.

** Starting with the **LA R12,48(,R15)** statement at offset +00002C, R12 points to offset +000030. This remains true until the **LM R14,R12,12(R13)** statement at offset +00004E.

4 The Set of Symbols Belonging to the csect named CODE

When it comes to using symbols, the important question is, “How far away is the symbol from a location pointed to by some register?” For the sample program, figure 4 shows the set of symbols belonging to the csect named CODE. It also shows the program offsets associated with each of those symbols, and it gives information about which registers point to what location, and when.

Using USINGs - What the Huh????

```
                                    76 ****************************************
                                    77 *                                      *
                                    78 * Sample Program -- With Program       *
                                    79 * Symbols Added, and Being Used.       *
                                    80 *                                      *
                                    81 ****************************************

000000             00000 000B0      83 CODE     CSECT ,
000000 47F0 F028          00028     84          B     ICATCHRZ-CODE(,R15)
000004 23                           85          DC    AL1(ICATCHRZ-ICATCHR)
000005 C485949640979996             86 ICATCHR  DC    C'Demo program for '
000016 E4A289958740E4E2             87          DC    C'Using USINGs class'
000028                              88 ICATCHRZ DS    0H

000028 90EC D00C          0000C     90          STM   R14,R12,12(R13)
00002C 41C0 F030          00030     91          LA    R12,BASE-CODE(,R15)
000030                              92 BASE     DS    0H

000030 4110 C034          00034     94          LA    R1,SAVEAREA-BASE(,R12)
000034 50D0 1004          00004     95          ST    R13,4(,R1)
000038 5010 D008          00008     96          ST    R1,8(,R13)
00003C 18D1                         97          LR    R13,R1

00003E 58F0 C028          00028     99          L     R15,FIRST#-BASE(,R12)
000042 5AF0 C02C          0002C    100          A     R15,SECOND#-BASE(,R12)
000046 50F0 C030          00030    101          ST    R15,ANSWER-BASE(,R12)

00004A 58D0 D004          00004    103          L     R13,4(,R13)
00004E 98EC D00C          0000C    104          LM    R14,R12,12(R13)
000052 17FF                        105          XR    R15,R15
000054 07FE                        106          BR    R14
000056 0000
000058 075BCD15                    108 FIRST#   DC    F'123456789'
00005C 3ADE68B1                    109 SECOND#  DC    F'987654321'
000060                             110 ANSWER   DS    F
000064                             111 SAVEAREA DS    XL72
                                   113          END   ,
```

5 Sample Program -- With Program Symbols Added and Being Used

Using the knowledge shown in figure 4, we can begin to put the program symbols to good use. This is shown in figure 5. Hardcoded offsets have been replaced by distance calculations. For example, the statement that had been coded as **`LA R1,52(,R12)`**, now reads **`LA SAVEAREA-BASE(,R12)`**. So now the assembler is doing the arithmetic, not us.

Let's look at figure 5 a little further. In most environments (well, many), the program can assume that R15 has been set to point to the program's first instruction (the “entry point address”). So:

- The **`B ICATCHRZ-CODE(,R15)`** instruction tells the assembler to compute the distance that ICATCHRZ is from the start of the program (28 bytes, hex), and then form an S-CON using that displacement, with R15 as a base (**`F028`**). At execution time, a jump to that location (ICATCHRZ) will occur.

- The **`LA R12,BASE-CODE(,R15)`** instruction tells the assembler to compute the distance that BASE is from the start of the program (30 bytes, hex), and then form an S-CON using that displacement, with R15 as a base (**`F030`**). At execution time, this will cause R12 to be loaded with the address of BASE.

- For the remainder of the program, R12 (not R15) will be used as the base register for references to instructions and data in the program. This is necessary because in almost all programs, R15 is used as a volatile work register, R12 is not. The sequence of statements L, A, and ST illustrate this.

The Basic USING Statement

The use of symbols in figure 5 allows the program to be considerably more flexible. The programmer does not have to manually calculate or recalculate the S-CONs, the assembler will do that automatically. However, it is still tedious! In each case, the programmer is telling the assembler what register to use as a base (R15 or R12 in this sample), and to what location that register points (CODE in the case of R15, and BASE in the case of R12).

An awful lot of typing can be saved, and the program can be made visually easier to read if there is a way for the assembler to somehow just “know” that at the start of the program R15 points to CODE, and later on that R12 points to BASE. THAT IS EXACTLY what the USING statement does!

Figure 6 shows the sample program with USINGs and DROPs added. The USING statement tells the assembler that it can **assume** that a given register points to a given location and that references to symbols following that location can be resolved using that register. The DROP statement tells the assembler that it can no longer make that assumption.

- The register given by the USING statement is called the "base register".
- The location given by USING statement is called the "base symbol" or the "base address".

Using USINGs - What the Huh????

```
                                        76 ****************************************
                                        77 *                                      *
                                        78 * Sample Program -- Using USINGs to    *
                                        79 * Simplify the Use of Symbols.         *
                                        80 *                                      *
                                        81 ****************************************

000000             00000 000B0          83 CODE     CSECT ,
             R:F   00000                 84          USING CODE,R15
000000 47F0 F028         00028          85          B     ICATCHRZ
000004 23                               86          DC    AL1(ICATCHRZ-ICATCHR)
000005 C485949640979996                 87 ICATCHR  DC    C'Demo program for '
000016 E4A289958740E4E2                 88          DC    C'Using USINGs class'
000028                                  89 ICATCHRZ DS    0H

000028 90EC D00C         0000C          91          STM   R14,R12,12(R13)
00002C 41C0 F030         00030          92          LA    R12,BASE
                                        93          DROP  R15
             R:C   00030                94          USING BASE,R12
000030                                  95 BASE     DS    0H

000030 4110 C034         00064          97          LA    R1,SAVEAREA
000034 50D0 1004         00004          98          ST    R13,4(,R1)
000038 5010 D008         00008          99          ST    R1,8(,R13)
00003C 18D1                            100          LR    R13,R1

00003E 58F0 C028         00058         102          L     R15,FIRST#
000042 5AF0 C02C         0005C         103          A     R15,SECOND#
000046 50F0 C030         00060         104          ST    R15,ANSWER

00004A 58D0 D004         00004         106          L     R13,4(,R13)
00004E 98EC D00C         0000C         107          LM    R14,R12,12(R13)
                                       108          DROP  R12
000052 17FF                            109          XR    R15,R15
000054 07FE                            110          BR    R14
000056 0000
000058 075BCD15                        112 FIRST#   DC    F'123456789'
00005C 3ADE68B1                        113 SECOND#  DC    F'987654321'
000060                                 114 ANSWER   DS    F
000064                                 115 SAVEAREA DS    XL72
                                       117          END   ,
```

6 Sample Program -- Using USINGs to Simplify the Use of Symbols.

There's something of an “honor system” here. If you use a USING statement telling the assembler that such-and-such register points to such-and-such location, then there had better be code that makes that assertion true. If it's not, then your program is likely to fail.

When you provide a USING statement, the assembler has no way of knowing whether the assertion being made by the USING is true or false. The assembler is only assembling your program, it is not running it. The loading of pointers into registers is something that happens at run time, not assembly time. On the other hand, the process of converting symbolic references into S-CONs is something that happens at assembly time, not run time. It is very important to keep clear in your mind that these are separate time frames and only loosely related events.

The coordination of the loading of registers with the presence of USING and DROP statements is something that you must do correctly. For example, in figure 6, in order for the **`USING BASE,R12`** statement to be correct, the preceding **`LA R12,BASE`** must be present. If it is not, the assembler is not going to complain, but your program will probably produce incorrect results.

Similarly, if a **`DROP R15`** were not issued prior to the **`L R15,FIRST#`** statement, then it might be possible (depending upon your program's particular logic) for the assembler to use R15 to form S-CONs even past the point where the correct value within R15 would be destroyed at run time.

The assembler has no way of knowing when any given register actually does or does not contain a particular value. The only thing it can do is trust your use of the USING and DROP statements. If you are wrong, then your program probably will fail. Generally, the USINGs should be placed only at those points in the code where the assumption would be true, and DROPs should be placed at the point where the assumption either is false or is no longer needed.

Using Dsects

At this point in our sample program, we've replaced several hardcoded S-CONs with symbols, but not all of them. Figure 7 shows those that remain. These all refer to fields in a control block known as the standard Register Save Area, or **RSA** for short. In this program, there's actually two RSA's, one provided by my caller (R13 starts out pointing to it), and one that I will provide to any subroutines that I might call (the field named SAVEAREA). Much of most programs' entry logic and return logic concerns itself with the use and manipulation of these two save areas.

Using USINGs - What the Huh????

```
                                        76 ****************************************
                                        77 *                                      *
                                        78 * Sample Program -- Using USINGs to    *
                                        79 * Simplify the Use of Symbols.         *
                                        80 *                                      *
                                        81 ****************************************

000000             00000 000B0          83 CODE     CSECT ,
             R:F   00000                 84          USING CODE,R15
000000 47F0 F028         00028          85          B     ICATCHRZ
000004 23                               86          DC    AL1(ICATCHRZ-ICATCHR)
000005 C485949640979996                 87 ICATCHR  DC    C'Demo program for '
000016 E4A289958740E4E2                 88          DC    C'Using USINGs class'
000028                                  89 ICATCHRZ DS    0H

000028 90EC D00C         0000C          91          STM   R14,R12,12(R13)
00002C 41C0 F030         00030          92          LA    R12,BASE
                                        93          DROP  R15
             R:C   00030                94          USING BASE,R12
000030                                  95 BASE     DS    0H

000030 4110 C034         00064          97          LA    R1,SAVEAREA
000034 50D0 1004         00004          98          ST    R13,4(,R1)
000038 5010 D008         00008          99          ST    R1,8(,R13)
00003C 18D1                            100          LR    R13,R1

00003E 58F0 C028         00058         102          L     R15,FIRST#
000042 5AF0 C02C         0005C         103          A     R15,SECOND#
000046 50F0 C030         00060         104          ST    R15,ANSWER

00004A 58D0 D004         00004         106          L     R13,4(,R13)
00004E 98EC D00C         0000C         107          LM    R14,R12,12(R13)
                                       108          DROP  R12
000052 17FF                            109          XR    R15,R15
000054 07FE                            110          BR    R14
000056 0000
000058 075BCD15                        112 FIRST#   DC    F'123456789'
00005C 3ADE68B1                        113 SECOND#  DC    F'987654321'
000060                                 114 ANSWER   DS    F
000064                                 115 SAVEAREA DS    XL72
                                       117          END   ,
```

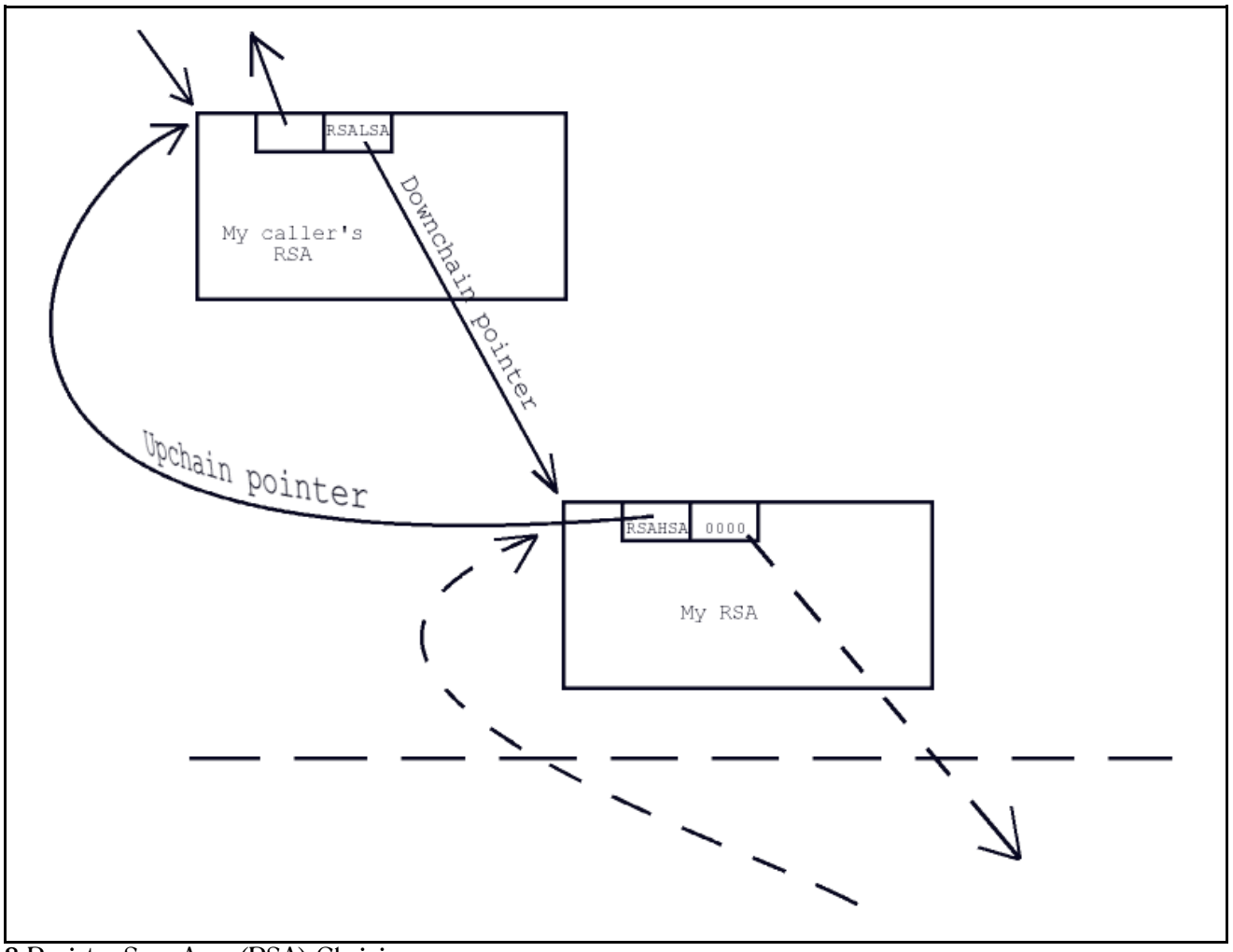
7 Sample Program -- Hardcoded S-CONs not yet converted to symbols

Register save areas, like any other typical control block, consist of a collection of fields, each with a defined purpose and each with a defined location in the control block. For example:

- The S-CON “12(R13)” refers to the field in which R14 is to be saved. (I call that field RSAR14.)
- The S-CONs “4(,R1)” and “4(,R13)” both refer to the field in an RSA that points to the next “higher” RSA, my caller's RSA in this case. (I call that field RSAHSA.)
- Similarly, the S-CON “8(,R13)” refers to the field that points to the next “lower” RSA. (I call it RSALSA.)

Using USINGs - What the Huh????

In any program, the local RSA is always referred to as being “lower” than the calling program's RSA, and as being “higher” than save areas belonging to subprograms that might be called. So my caller's RSALSA field will point to my save area, and my save area's RSAHSA field will point back to my caller's RSA. This is called “cross chaining the save areas”. (See the diagram in figure 8.) The opening logic of the sample program cross chains my save area with my callers, and if I call any subroutines, then (one at a time) their opening logic will cross chain their save areas with mine.



8 Register Save Area (RSA) Chaining

Anyway, wouldn't it be nice if there were a way to define a set of symbols that matched the layout of an RSA? And then we could use those symbols in the code instead of having to use hardcoded displacements? Ok, check out figure 9. This shows a dsect containing a set of symbols corresponding to the fields in an RSA.

Using USINGs - What the Huh????

Symbol Sets, an Important Concept

The assembler organizes symbols into distinct sets. Generally, a “symbol set” is the set of all symbols belonging to a particular csect (code section) or dsect (dummy section). Each symbol in the set is associated with an “offset” from the start of the set. The important relationship between symbols is their distances from each other. For example, if one symbol has an offset of +03C and another has an offset of +238, then the distance between them is 1FC.

A USING statement creates “addressability” for some or all symbols in one set. The USING statement generally names one of the symbols as being the base symbol, and declares that a particular register (to be called the “base register”) points to a storage location to be represented by that symbol. Addressability to other symbols in the set is then computed in terms of the distances of those symbols from the base symbol. Thus, the other symbols are made to represent locations following the location pointed to by the base register.

To obtain addressability to symbols in a different set, it is necessary to issue another USING, referencing some offset within that other set.

```
                          46 ****************************************
                          47 ** RSA      = STANDARD REGISTER SAVE AREA*
                          48 ** RSA      = DSECT AND BASE             *
                          49 ****************************************
                          52 RSA      #DSA  ,
000000      00000 00048   54+RSA      DSECT 0D   STD REGISTER SAVE AREA
000000                    55+RSAWD1   DS    A    WORD-1 (UNUSED)
000004                    56+RSAHSA   DS    A    HIGHER SAVE AREA PTR
000008                    57+RSALSA   DS    A    LOWER SAVE AREA PTR
00000C                    58+RSAR14   DS    A    (RETURN ADDRESS)
000010                    59+RSAR15   DS    A    (ENTRY POINT)
000014                    60+RSAR0    DS    A
000018                    61+RSAR1    DS    A
00001C                    62+RSAR2    DS    A
000020                    63+RSAR3    DS    A
000024                    64+RSAR4    DS    A
000028                    65+RSAR5    DS    A
00002C                    66+RSAR6    DS    A
000030                    67+RSAR7    DS    A
000034                    68+RSAR8    DS    A
000038                    69+RSAR9    DS    A
00003C                    70+RSAR10   DS    A
000040                    71+RSAR11   DS    A
000044                    72+RSAR12   DS    A
000048                    74+RSAZ     DS    0X   Z'RSA
            00048         75+RSAL     EQU   RSAZ-RSA L'RSA
```

9 A dsect mapping the fields in an standard Register Save Area

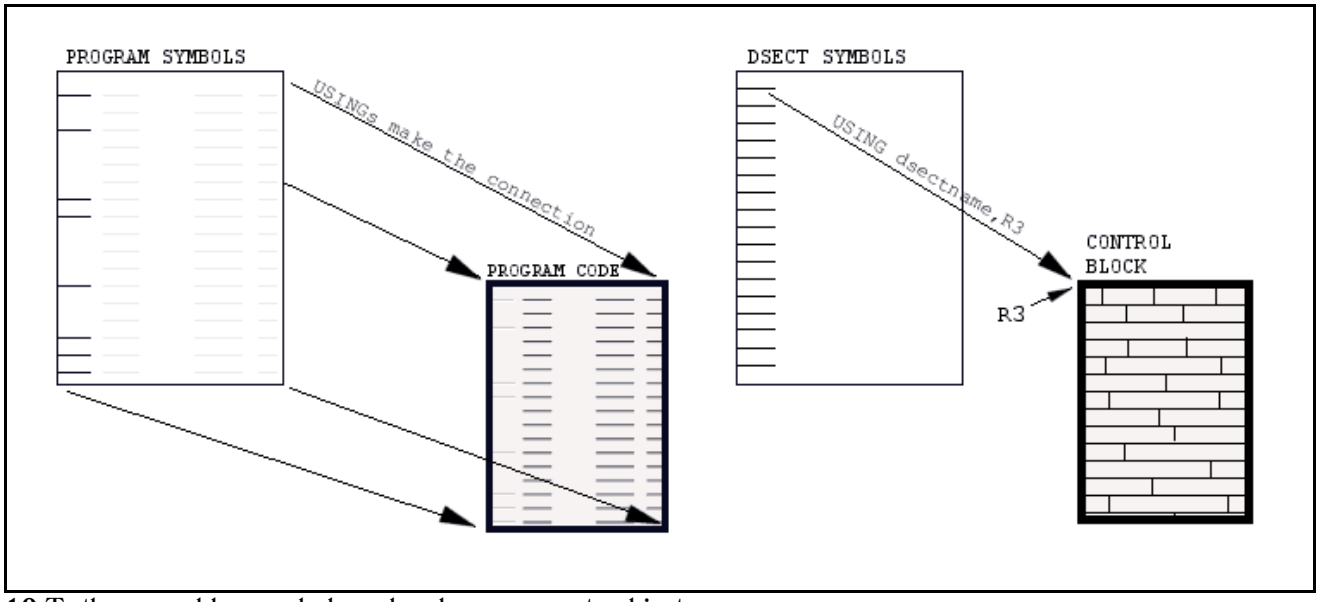
What's a “dee sect”? Well, dsects are nothing more than named collections of symbols that are not associated with generated code. They are not associated with code because they are often used to reference things that are built dynamically or that exist outside of the program.

When you write a program, your generally write a bunch of machine instructions and a bunch of data fields that both refer to and are labelled by a bunch of symbols. It is important to realize that there are two structures here:

Using USINGs - What the Huh????

There is the code, and there is the set of symbols that corresponds to that code. (See figure **10**.) In order to understand USINGs, it is very important to keep this thought clear in your mind: The set of symbols is something that may correspond to your code, but it remains a distinct and separate entity from your code.

The code is a runtime construct. It is what is loaded into storage and executed. The symbol set is an assembly time construct. It is a tool that the assembler uses to resolve references in your code to locations either within or outside of your code.



10 To the assembler, symbols and code are separate objects

When you issue a USING to make references resolvable, you are, in effect, performing an assembly time assignment of the symbols to the code. You could issue a different USING and thereby tell the assembler to assign the symbols to some other object that's not your code. Usually, this is not a useful thing to do, but you could do it. The assembler will be obedient. It won't complain. But your program, when executed, will fail miserably.

A dsect is a collection of symbols that does not correspond to generated code. Usually, you create dsects to represent objects other than code, such as control blocks and other data areas. When you've loaded a register to point to such an object, you can then issue a USING to notify the assembler that you intend to use the dsect's symbols to represent the fields within that object.

Getting Back to Using Dsects

To make use of the symbols in the RSA dsect, we simply have to proceed as we did earlier. First, let's replace the hardcoded displacements shown back in figure **7**, with the assembler calculated displacements shown in figure **11**.

Using USINGs - What the Huh????

```
                                        76 ****************************************
                                        77 *                                      *
                                        78 * Sample Program -- Using Symbols in   *
                                        79 * Control Blocks                       *
                                        80 *                                      *
                                        81 ****************************************

000000             00000 000B0          83 CODE     CSECT ,
             R:F   00000                 84          USING CODE,R15
000000 47F0 F028         00028          85          B     ICATCHRZ
000004 23                               86          DC    AL1(ICATCHRZ-ICATCHR)
000005 C485949640979996                 87 ICATCHR  DC    C'Demo program for '
000016 E4A289958740E4E2                 88          DC    C'Using USINGs class'
000028                                  89 ICATCHRZ DS    0H

000028 90EC D00C         0000C          91          STM   R14,R12,RSAR14-RSA(R13)
00002C 41C0 F030         00030          92          LA    R12,BASE
                                        93          DROP  R15
             R:C   00030                94          USING BASE,R12
000030                                  95 BASE     DS    0H

000030 4110 C034         00064          97          LA    R1,SAVEAREA
000034 50D0 1004         00004          98          ST    R13,RSAHSA-RSA(,R1)
000038 5010 D008         00008          99          ST    R1,RSALSA-RSA(,R13)
00003C 18D1                            100          LR    R13,R1

00003E 58F0 C028         00058         102          L     R15,FIRST#
000042 5AF0 C02C         0005C         103          A     R15,SECOND#
000046 50F0 C030         00060         104          ST    R15,ANSWER

00004A 58D0 D004         00004         106          L     R13,RSAHSA-RSA(,R13)
00004E 98EC D00C         0000C         107          LM    R14,R12,RSAR14-RSA(R13)
                                       108          DROP  R12
000052 17FF                            109          XR    R15,R15
000054 07FE                            110          BR    R14
000056 0000
000058 075BCD15                        112 FIRST#   DC    F'123456789'
00005C 3ADE68B1                        113 SECOND#  DC    F'987654321'
000060                                 114 ANSWER   DS    F
000064                                 115 SAVEAREA DS    XL(RSAL)
                                       117          END   ,
```

11 Sample Program -- Using Symbols in Control Blocks

Next, I want to add USINGs so that I don't have to be manually typing the subtractions all the time. But now I've got a new problem: The code is dealing with two RSAs, not just one. So I have to decide to which RSA I'm going to assign the symbols and when I'm going to make the assignment. This is shown in figure **12**. Let's examine this puppy closely:

Using USINGs - What the Huh????

```
                                        76 ****************************************
                                        77 *                                      *
                                        78 * Sample Program -- Using USINGs for   *
                                        79 * Control Blocks                       *
                                        80 *                                      *
                                        81 ****************************************

000000             00000 000B0          83 CODE     CSECT ,
             R:D   00000                 84          USING RSA,R13
             R:F   00000                 85          USING CODE,R15
000000 47F0 F028          00028         86          B     ICATCHRZ
000004 23                               87          DC    AL1(ICATCHRZ-ICATCHR)
000005 C485949640979996                 88 ICATCHR  DC    C'Demo program for '
000016 E4A289958740E4E2                 89          DC    C'Using USINGs class'
000028                                  90 ICATCHRZ DS    0H

000028 90EC D00C          0000C         92          STM   R14,R12,RSAR14
00002C 41C0 F030          00030         93          LA    R12,BASE
                                        94          DROP  R15
             R:C   00030                95          USING BASE,R12
000030                                  96 BASE     DS    0H

000030 4110 C034          00064         98          LA    R1,SAVEAREA
000034 50D0 1004          00004         99          ST    R13,RSAHSA-RSA(,R1)
000038 5010 D008          00008        100          ST    R1,RSALSA
00003C 18D1                            101          LR    R13,R1

00003E 58F0 C028          00058        103          L     R15,FIRST#
000042 5AF0 C02C          0005C        104          A     R15,SECOND#
000046 50F0 C030          00060        105          ST    R15,ANSWER

00004A 58D0 D004          00004        107          L     R13,RSAHSA
00004E 98EC D00C          0000C        108          LM    R14,R12,RSAR14
                                       109          DROP  R12
000052 17FF                            110          XR    R15,R15
000054 07FE                            111          BR    R14
                                       112          DROP  R13
000056 0000
000058 075BCD15                        114 FIRST#  DC    F'123456789'
00005C 3ADE68B1                        115 SECOND# DC    F'987654321'
000060                                 116 ANSWER  DS    F
000064                                 117 SAVEAREA DS   XL(RSAL)
                                       119          END   ,
```

12 Sample Program -- Using USINGs for Control Blocks

- At line #84, I've inserted a **`USING RSA,R13`** statement. This documents that at program entry time R13 has been set by my caller to point to a register save area that he has created and that he is making available for my use. This USING statement causes the RSA symbols set to be "mapped" to the storage pointed to by R13. The "base" of this mapping is the symbol named "RSA". After this USING statement, if the assembler encounters a reference to a symbol from the RSA set (and if that reference does not already have

a basing symbol subtracted from it), then it will resolve that reference using R13 as the base register and "RSA" as the base symbol.

- At line #92 the store multiple has been changed to **`STM R14,R12,RSAR14`**. (It used to be "**`STM R14,R12,RSAR14-RSA(R13)`**".) The assembler has used information from the USING statement to process the "RSAR14" reference as if "RSAR14-RSA(R13)" had been given.

- At line #99, I have not been able to change "**`...RSAHSA-RSA(,R1)`**"! This is because R1 does not point to the same save area as R13, and the USING statement has told the assembler to use R13 as the base for RSA references, not R1. So I've had to leave #99 as a fully written out S-CON.

- Note the **`LR R13,R1`** at line #101. This is causing R13 to be changed from pointing to my caller's save area to pointing to my local save area (at SAVEAREA). From now on when I refer to a symbol in the RSA set, I'll be referring to my local save area, not my caller's save area. Well, since the RSA symbols will map my local save area just as correctly as it did my caller's save area, I don't need to notify the assembler of this change. I don't need to issue any DROPs and USINGs here.

- Now we get to line #107: **`L R13,RSAHSA`**. Up until this statement is executed, R13 points to my local save area. Upon executing this statement, the contents of the RSAHSA field are loaded into R13, thus changing R13 from pointing to my local save area to pointing again to my caller's save area. Again, I don't need to issue a DROP and USING here because the RSA dsect will just as correctly map the caller's RSA as it did my local save area.

- The next statement (**`LM R14,R12,RSAR14`**) restores, from my caller's save area, my caller's registers that I had previously save therein.

Using Named USINGs

There are some problems with the above. First, one symbol set is being used to map two instances of a register save area. Just which RSA is being mapped depends upon where, in the code, the reference occurs. This works, but its confusing. Unless you examine the logic very carefully, it may not be real clear which RSA is being referenced at what point in time and even that there are two RSAs involved!

Second, there is a point in the code (lines 98, 99, and 100) where both register save areas are being processed at the same time, but we've been able to assign the RSA symbol set to only one of those instances. What is needed is an unambiguous way to use the same symbol set twice, simultaneously. (Named USINGs!)

Using USINGs - What the Huh????

```
                                        76 ****************************************
                                        77 *                                      *
                                        78 * Sample Program -- Using named USINGs *
                                        79 * for Multiple Instances of a Control  *
                                        80 * Block                                *
                                        81 *                                      *
                                        82 ****************************************

000000             00000 000B0          84 CODE     CSECT ,
             R:D   00000                 85 HI       USING RSA,R13
             R:F   00000                 86          USING CODE,R15
000000 47F0 F028         00028          87          B     ICATCHRZ
000004 23                               88          DC    AL1(ICATCHRZ-ICATCHR)
000005 C485949640979996                 89 ICATCHR  DC    C'Demo program for '
000016 E4A289958740E4E2                 90          DC    C'Using USINGs class'
000028                                  91 ICATCHRZ DS    0H

000028 90EC D00C         0000C          93          STM   R14,R12,HI.RSAR14
00002C 41C0 F030         00030          94          LA    R12,BASE
                                        95          DROP  R15
             R:C   00030                96          USING BASE,R12
000030                                  97 BASE     DS    0H

000030 4110 C034         00064          99          LA    R1,SAVEAREA
             R:1   00000                100 LO       USING RSA,R1
000034 50D0 1004         00004          101         ST    R13,LO.RSAHSA
000038 5010 D008         00008          102         ST    R1,HI.RSALSA
00003C 18D1                             103         LR    R13,R1
                                        104         DROP  LO
                                        105         DROP  HI
             R:D   00000                106         USING RSA,R13

00003E 58F0 C028         00058          108         L     R15,FIRST#
000042 5AF0 C02C         0005C          109         A     R15,SECOND#
000046 50F0 C030         00060          110         ST    R15,ANSWER

00004A 58D0 D004         00004          112         L     R13,RSAHSA
                                        113         DROP  R13
             R:D   00000                114 HI       USING RSA,R13
00004E 98EC D00C         0000C          115         LM    R14,R12,HI.RSAR14
                                        116         DROP  R12
000052 17FF                             117         XR    R15,R15
000054 07FE                             118         BR    R14
                                        119         DROP  HI
000056 0000
000058 075BCD15                         121 FIRST#  DC    F'123456789'
00005C 3ADE68B1                         122 SECOND# DC    F'987654321'
000060                                  123 ANSWER  DS    F
000064                                  124 SAVEAREA DS   XL(RSAL)
                                        126         END   ,
```

13 Sample Program -- Using named USINGs for Multiple Instances of a Control Block

See figure 13. The assembler allows you to put names onto USING statements. Without names, when two USINGs reference the same base name in the same symbol set, one will effectively override the other. But when names are used, both USINGs will coexist without interfering with each other.

Let's look at figure 13 carefully:

- The name **HI** has been added to the USING statement at line #85. Then the references to the RSA symbol set at lines #93 and #102 have been "qualified" by prefixing them with "**HI.**" (H I dot). This causes these references to be resolved using the base register (R13) and the symbol base (RSA) declared by the USING statement labelled **HI**.

- A **LO USING RSA,R1** statement has been added at line #100, following the **LA R1,SAVEAREA**. Then a "**LO.**" has been added to the reference in line #101. This qualification causes that reference to be resolved using the base register (R1) and the symbol base (RSA) declared by the USING statement labelled **LO**.

- Once the cross chaining is finished, the reference to the "higher register save area" and the concept of a "lower register save area" are no longer needed, so DROPs are issued at lines #104 and #105. Notice that these DROPs are "by name", not "by register". The rules are:
  - Whenever a named USING is being dropped, the DROP must refer to that USING by its name, not its register.
  - Whenever an unnamed USING is being dropped, if that USING is register based[1], then the DROP must refer to that USING by the register.

- Even though cross chaining is finished, R13 still points to my local save area, and it will remain so pointed for the rest of the program. To document this fact, an unnamed **USING RSA,R13** is issued at line #106.

- Return linkage starts at line #112. In preparation for the return, the **L R13,RSAHSA** changes R13 from pointing to my local save area to pointing back again to my caller's save area. To document that fact, the **DROP R13** statement (#113) cancels the declaration that R13 points to my local save area, and the **HI USING RSA,R13** statement (#114) resumes the declaration that R13 now points to my caller's save area (the immediately higher save area).

- The **LM R14,R12,HI.RSAR14** statement (#115) then restores my caller's registers from the higher save area.

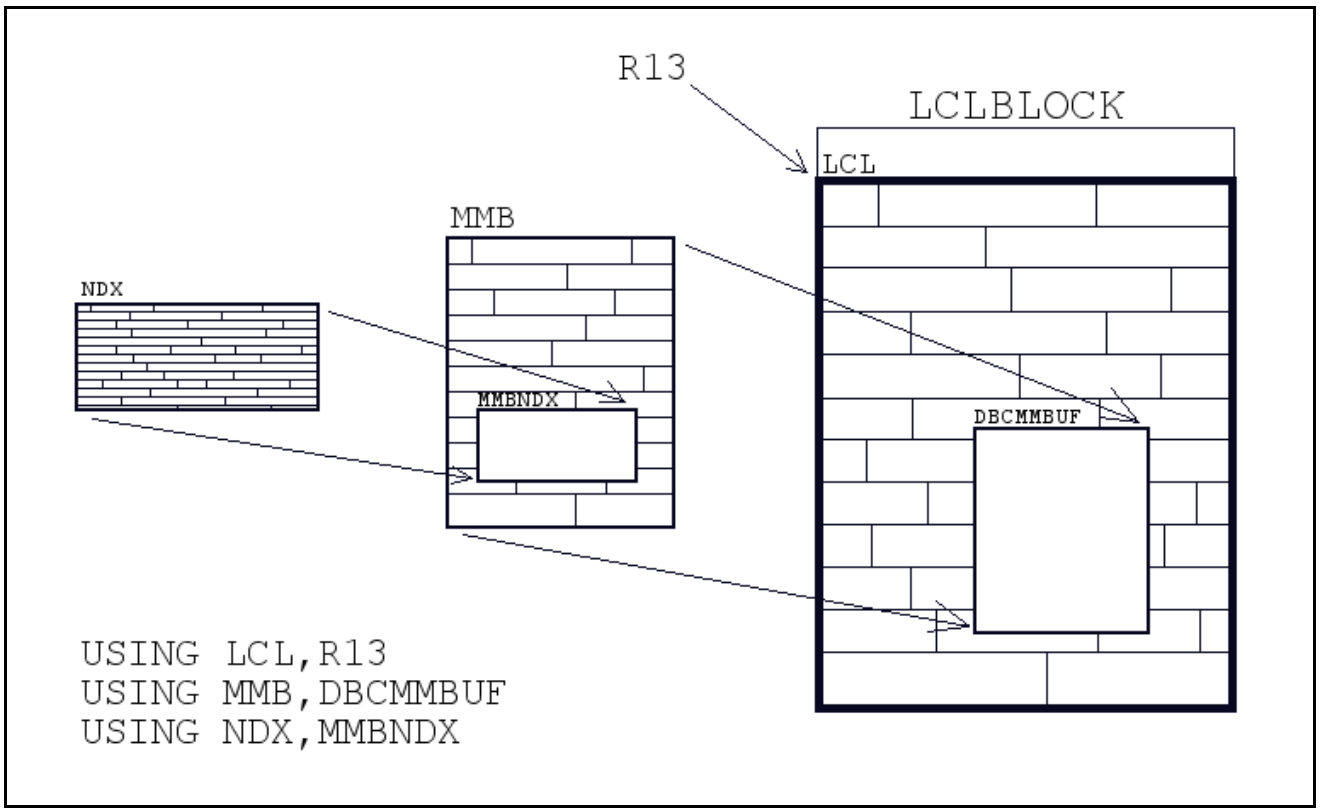
Strictly speaking, the transition from the unnamed USING to the HI-named USING (lines #113 and #114) is not necessary. The returned linkage logic would have worked perfectly well without making that change. It's just my opinion that making that transition clarifies the logic and makes the code just a little bit easier to understand.

Using Dependent USINGs

Cole Software's primary line of products is XDC®. The following examples are taken from our 64-bit version named z/XDC®.

[1] Unnamed field based USINGs cannot be explicitly dropped. (A discussion of USINGs that are based upon fields instead of registers is coming up.)

"Dependent" USINGs are USINGs that base the resolution of one symbol set upon a symbol from another set. In other words, the symbols from the dependent set are assigned to locations within the primary set, and the base register used to resolve the dependent symbols will be the same register used to resolve the primary symbols.



14 Assigning one symbol set within another within another

The diagram in figure 14 illustrates this. Within z/XDC®, there exists a control block named the MMB. (Its dsect map is shown in figure 15.) Generally, whenever I need to access an MMB, I'll call a service routine that will locate the particular MMB that I need and make a copy of it in an extraction buffer named DBCMMBUF. This buffer is located within a data area named LCLBLOCK (figure 16).

Also, the MMB itself contains another control block called the NDX (figure 17). With suitable dependent USINGs, R13 can be used to resolve symbols in both the LCLBLOCK data area as well as the MMB within, and the NDX within that.

```
000000      000000 000CC 3400+MMB      DSECT ,
000000                   3401+MMBCHAIN DS    A             --> NEXT MMB
000004                   3402+MMBNAME  DS    CL(LABLMAXL)  MODULE NAME
000043                   3403+MMBMAJNM DS    CL(LABLMAXL)  MAJOR NAME
000082                   3405+MMBASNME DS    CL8           ADDRESS SPACE
                         3408+*                            ENTRY ADDRESS
000090                   3410+MMBEAD64 DS    0AD           ADDRESS 64-BIT
000090 00000000          3411+MMBEA_HI DC    A(0)          ADDRESS HI-HALF
000094 00000000          3412+MMBEA_LO DC    A(0)          ADDRESS LO-HALF
                                       ...
0000AC                   3439+         DS    0F          |
0000AC                   3440+MMBNDX   DS    XL(NDXLEN)  V NAMES INDEX
            0000CC       3442+MMBEND   DS    0X            Z'MMB
            0000CC       3443+MMBLEN   EQU   MMBEND-MMB    L'MMB
```

15 MMB dsect map

To reference the fields within the LCLBLOCK, I issue a normal USING statement: **`USING LCL,R13`**. This means that all fields within the LCLBLOCK dsect, at and following the LCL label, will be resolvable using R13 as the base register and LCL as the base symbol. (All labels preceding LCL will not be resolvable.)

```
000000      000000 004000 8718+LCLBLOCK DSECT ,
000000                    8725+LCLTWBH0 DS    XL(TWBHL)  TWA BLOCK HEADER
000010                    8733+LCL      DS    0D         STD REG. SVAREA
                                        ...
0007F0                    9939+         DS    0F
0007F0                    9940+DBCMMBUF DS    XL(MMBLEN) MMB BUFFER
```

16 MMB buffer with the LCLBLOCK dsect

```
000000      000000 000020 3376+NDX      DSECT ,
000000                    3377+NDXBYNME DS    A    | @'INDEX SORTED BY NAME
000004                    3378+NDXSZNME DS    F    V L'NAME ORDERED INDEX
000008                    3379+NDXBYADR DS    A    | @'INDEX SORTED BY ADDR
00000C                    3380+NDXSZADR DS    F    V L'ADDR ORDERED INDEX
000010                    3381+NDXBYSEQ DS    A    | @'SEQ. ORDERED INDEX
000014                    3382+NDXSZSEQ DS    F    V L'SEQ. ORDERED INDEX
000018                    3383+NDXMAPST DS    A    | @'MAP DATA HEAP AND
                          3384+*                   | HEADER FIELDS (DWORD
                          3385+*                   | ALIGNED)
00001C                    3386+NDXMAPND DS    A    V END OF MAP DATA
            000020        3387+NDXEND   DS    0X
            000020        3388+NDXLEN   EQU   NDXEND-NDX
```

17 NDX dsect map

Using USINGs - What the Huh????

When I've copied an MMB into DBCMMBUF and I want to reference the fields of that copy, I'll issue the statement: **`USING MMB,DBCMMBUF`**. This assigns the MMB symbol set to represent locations within DBCMMBUF. Similarly, I'll then issue a **`USING NDX,MMBNDX`** in order to reference the NDX within the MMBNDX buffer.

Figure 18 shows a snippet of code that makes good use of dependent USINGs. A subroutine (DBCADCSE) is called to attempt to locate a csect map. If it's successful (+8 return), then an MMB is returned in the DBCMMBUF buffer:

- The **`USING MMB,DBCMMBUF`** statement assigns the MMB dsect to the DBCMMBUF buffer, and the **`USING NDX,MMBNDX`** statement assigns the NDX dsect to the MMBNDX field within the MMB control block.

```
0692 4DE0 C0DE          000DE 6833    BAS    R14,DBCADCSE  GET CSECT/ESD INFO
0696 47F0 88E6          008E6 6834    B      DIMODONE      +0 NOT MAPPED; SKIP
069A 47F0 88CA          008CA 6835    B      DIOVMISS      +4 MAP'D BUT CSECT
*                                                              NOT LOADED
                                      6836    PUSH  USING          +8 GOT CSECT + MMB
           D 7E0 00000 007F0 6837    USING MMB,DBCMMBUF  DCL MMB FIELDS
           D 88C 00000 000AC 6838    USING NDX,MMBNDX    DCL NDX FIELDS
                                             ...
06CE 5800 D894          00008 6869    L      R0,NDXBYADR   @'ADDR-ORDERED
*                                                          INDEX OF XTLs
06D2 5E00 D898          0000C 6871    AL     R0,NDXSZADR   Z'INDEX OF XTLs
                                      6872    POP   USING          DONE: MMB + ITS NDX
```

18 Using an NDX within an MMB within the LCL

- The **`PUSH USING`** and **`POP USING`** statements are for dropping the MMB and NDX symbols when they are no longer needed. This is because unnamed dependent USINGs cannot be explicitly dropped.[1]

PUSH/POP USINGs

The set of USINGs in effect at a given point in the code is called the "USING environment". Every time you issue a USING or DROP statement, you are changing the USING environment.

Sometimes it is useful to be able to change the USING environment, then restore it later without having to know explicitly what the original USING environment was. The PUSH USING and POP USING statements make this possible.

[1] More about this in a moment.

Using USINGs - What the Huh????

Example: Within z/XDC®, subroutines generally are self-contained, self-sufficient objects. They have their own environments (including the USING environment) that is independent of other code coming before or afterwards. Accordingly, a typical subroutine will start with a whole bunch of setup statements as shown in figure 19:

```
*************************************************************** 07/98 X34
* Local environment setup.                                     * 07/98 X34
*************************************************************** 07/98 X34
          PUSH  USING                SAVE THE UNKNOWN           05/98 X34
          DROP  ,                    KILL THE UNKNOWN           05/98 X34
          USING LCL,LCLREG           DCL LCL BASE               05/98 X34
          USING BVT,BVTREG           DCL BVT BASE               05/98 X34
          USING RSTK,RSTKREG         DCL CURRENT RSTK BASE      05/98 X34
          USING ACSERSTK,RSTKWORK    DCL WORK FIELDS OVERLAY    07/98 X34
          USING #DBCADCS,PGMBASE1    LOCAL CODE BASE                  X14
          ENTRY #DBCADCS                                        09/89 X21
#DBCADCS  DS    0H                                              05/98 X34
          ...
          B     QUICKR04             RETURN +4 (SOSO)           05/98 X34
          POP   USING                KILL LOCAL DECLARES        05/98 X34
          LTORG ,                    LOCAL LITERALS                   X14
                =A(CMPRAD64)
                =Y(MIXL)
```

19 PUSH/POP USING example

- The **PUSH USING** statement saves the current USING environment, whatever it is.
- The **DROP ,** statement drops the entire using environment, no matter what it is. This creates a known state: No USINGs defined, whatsoever.
- Then the various USING statements define the environment needed by the subroutine.
- At the end of the subroutine, the **POP USING** statement cancels all of the USINGs that were declared in the subroutine and that remain outstanding. It then restores the external USING environment, whatever it was.

Dropping Dependent USINGs

Dependent USINGs, like register based USINGs, can be either named or unnamed. A named dependent USING can be dropped the same as a named register based USING: by name. Example:

```
         USING LCL,R13          DECLARE THE PRIMARY SYMBOL SET
N        USING MMB,DBCMMBUF     DECLARE THE DEPENDENT SYMBOL SET
         ...
         DROP  N                DONE WITH THE DEPENDENT SYMBOL SET
```

Unnamed dependent USINGs, on the other hand, cannot be explicitly dropped. This is because the DROP statement permits dropping a USING only by name or by register, but an unnamed dependent USING has neither a name nor an explicit or unique reference to a register. There simply is no syntax available for a DROP statement to refer specifically to an unnamed dependent USING.

Using USINGs - What the Huh????

So there's no direct way to drop an unnamed dependent USING, but there are several indirect ways to do so. They are illustrated in figure 20. One such is to bracket, with a **PUSH/POP USING**, the range of code in which the dependent USING is needed. The **PUSH USING** causes the assembler to save the current set of USING declarations, and the corresponding **POP USING** causes that saved set to be restored. All USINGs issued between the two statements, including unnamed dependent USINGs, are dropped by the POP.

Caveat: Be careful of "container violations". In a sense, a PUSH/POP sequence forms a "container": The PUSH saves a state, the POP restores that state. Anything that happens to that state, within the PUSH/POP range, is contained. The problem with using a PUSH/POP for dropping unnamed dependent USINGs, is that the POP may effectively restore USINGs you thought you had dropped and drop other USINGs that you didn't want to drop yet.

Using **PUSH/POP**:

```
          PUSH  USING           Save current declares
          USING MMB,DBCMMBUF    Declare MMB to be dependent upon LCL/DBCMMBUF
          L     R1,NDXBYADR     @'address ordered index
          ...
          POP   USING           Restore prior USINGs
```

Using a named **USING**:

```
B         USING MMB,DBCMMBUF    Declare MMB to be dependent upon LCL/DBCMMBUF
          L     R1,B.NDXBYADR   @'address ordered index
          ...
          DROP  B               Done w/DBCMMBUF
```

Using **DROP** (not recommended):

```
          USING MMB,DBCMMBUF    Declare MMB to be dependent upon LCL/DBCMMBUF
          L     R1,NDXBYADR     @'address ordered index
          ...
          DROP  R13             Kill the primary base and all dependents
          USING LCL,R13         Restore only the primary base
```

20 Strategies for dropping dependent USINGs

Examples of both these problems are shown in figure 21: The domain[1] of the DMY USING overlaps the start of the PUSH/POP range, and the domain of the GBL USING overlaps the end of the range. So when the POP is issued:

- The DMY USING is unexpectedly restored and has to be dropped again.
- The GBL USING is unexpectedly lost and has to be declared again.

[1] The "domain" of a USING is the range of code during which the USING is in effect. It starts with the USING statement, and ends with the corresponding DROP (or POP) statement.

```
         USING DMY,R2           DCL BASE FOR DSECT CONTROLS
         ...
         PUSH  USING            SAVE CURRENT DECLARES
         USING MMB,DBCMMBUF     DECLARE MMB TO BE DEPENDENT UPON LCL/DBCMMBUF
         L     R1,NDXBYADR      @'ADDRESS ORDERED INDEX
         ...
         DROP  R2               DONE W/DMY BASE
         ...
         L     R11,LCLGBLAD     @'XDC-GBL
         USING GBL,R11          DCL ITS BASE
         ...
         POP   USING            RESTORE PRIOR USINGS
         DROP  R2               RE-KILL DMY BASE
         USING GBL,R11          RESTORE XDC-GBL BASE
         ...
         LA    R1,GBLTRAQH-(TRCBNEXT-(TRACECB) LOAD TRACECB QUEUE SCANNER
```

21 USINGs whose domains overlap PUSH/POP boundaries

Ok, referring again to figure **20**, another way to drop an “unnamed” dependent USING is to name it. Then it can be dropped by name.

Still another way is to drop the register on which the dependent USING is based. When any USING is dropped, all other USINGs that were dependent upon that USING are also dropped.

The Range of a USING

The "range" of a USING is the range of offsets, within a symbol set, that can be resolved as a result of the USING. Symbols falling within that range are resolvable. Symbols falling outside of that range cannot be resolved.

Normally, the range of a USING is 4096 bytes. This is because the USING statement is simply a way to automate the creation of S-CONs. As noted earlier, an S-CON is a 2-byte value that may be found in either the 2nd or 3rd halfword of a machine instruction. (See figure **2**.) Its first digit identifies the register being used as a base, and the remaining three digits identify the displacement value to be added to the contents of the register to form the address to be referenced.

Note that the displacement is only three hex digits wide. This means that the displacement can range in value from X'000' to X'FFF' (4095 in decimal). Consequently, this limits the maximum range of a USING.

Within a symbol set, the range of a USING starts at the base symbol referenced by the USING statement, and continues through 4095 bytes past that symbol. This means that all symbols falling within the USING's range are resolvable through use of the base register specified on the USING statement. But all symbols coming before the base symbol, as well as all symbols coming at and after the base symbol+4096, are not resolvable.

Using USINGs - What the Huh????

Usually for dsects, the name of the dsect (i.e. the start of the dsect) is used as the base symbol, but not always. The standard mapping macros for the CVT[1], the TCB[2], and the RB[3] are good examples of exceptions. All of these control blocks have prefix sections (fields that occur prior to the field normally pointed to from other control blocks).

Figure 22 shows the case of the CVT. A fundamental and well known fact about MVS[4] is that the hardcoded location X'00000010' always points to the CVT. What is less well known is that it does not actually point to the start of the CVT, it points 256 bytes into the CVT. Those skipped-over 256 bytes constitute the CVT's "prefix section".

```
                                  ******************************************
                                  *        Communications Vector Table     *
                                  ******************************************
                          491+          CVT   DSECT=YES,PREFIX=YES,LIST=YES
             000010       679+CVTPTR    EQU   16 -           ABSOLUTE ADDRES
                          680+*                              OF THE STANDARD
                          681+*                              CVT POINTER.
                          682+*
                          681+*                    BEGINNING OF CVT PREFIX
000000       000000 000600 683+CVTFIX    DSECT -             CVTMAP-256 - PR
000000                     684+           DS    CL216 -       RESERVED
0000D8                     685+CVTPROD    DS    0CL16 -       SYSTEM CONTROL
0000D8 4040404040404040    686+CVTPRODN   DC    CL8'        ' PRODUCT NAME OF
0000E0 4040404040404040    693+CVTPRODI   DC    CL8'        ' PRODUCT FMID ID
                          694+*                              CONTROL PROGRAM
                          697+CVTVERID   DC    CL16' '       OPTIONAL USER P
0000F8                     699+           DS    H -           RESERVED
0000FA                     700+CVTMDL     DS    CL2 -         CPU NUMBER IN S
0000FC                     703+CVTRELNO   DS    0CL4 -        RELEASE NUMBER
0000FC                     704+CVTNUMB    DS    CL2 -         RELEASE NUMBER
0000FE                     705+CVTLEVL    DS    CL2 -         LEVEL NUMBER OF
                          707+*                    END OF CVT PREFIX
                          708+*
                          712+*                    START OF CVT
000100                     713+CVTMAP     DS    0D
000100 00000000            715+CVTTCBP    DC    V(IEATCBP) -  ADDRESS OF PSAT
000104 00000000            716+CVT0EF00   DC    V(IEA0EF00) - ADDRESS Of ROUT
```

22 CVT prefix fields

Typically when writing code to reference the CVT, one would load the CVT's address into a register, and then issue a USING statement to notify the assembler that references to CVT fields can be resolved using that register. Example:

```
L     R2,CVTPTR
USING CVTTCBP,R2
```

[1]Communications Vector Table

[2]Task Control Block

[3]Request Block

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

CVTTCBP is the name of the CVT's first field located past the prefix section. So all CVT fields located at and past the CVTTCBP field are resolvable using R2 so long as none of those fields are further away from CVTTCBP than 4095 bytes.[1] In other words, all symbols in the CVT's dsect, starting between +X'000100' and +X'0010FF', are resolvable. All symbols starting between +X'000000' and +X'0000FF' (i.e. the prefix fields) are not resolvable.

If you wanted to access the CVT's prefix fields, you would have to write code such as this:

```
L     R2,CVTPTR
SH    R2,=Y(CVTTCBP-CVTFIX)
USING CVTFIX,R2
```

Extending the Range of a USING[2]

Sometimes, 4096 bytes is not enough. You could (heaven help us) create a control block that is longer than 4096 bytes, but more typically, you might write a program that is too long. When this happens, all symbols located more than 4095 bytes past the program's base symbol will not be resolvable. Not resolvable unless you do something such as the following:

```
MYSUB    DS    0H                    ENTRY ADDRESS
         BASR  R12,0                 LOAD A LOCAL BASE
MYBASE   DS    0H                    NAME IT
         LA    R15,1                 LOAD AN INCREMENT
         LA    R11,X'FFF'(R15,R12)   LOAD A 2ND BASE REGISTER
         LA    R8,X'FFF'(R15,R11)    LOAD A 3RD BASE REGISTER
         USING MYBASE,R12,R11,R8     DECLARE 3 CODE SECTION BASES
```

This little bit of code does the following:

- It loads R12 with the intended base address: MYBASE.
- It sets R11 to +X'001000' past the location pointed to by R12.
- It sets R8 to +X'001000' past the location pointed to by R11.
- The USING statement is a shorthand way of telling the assembler that:
  - R12 points to MYBASE.
  - R11 points to MYBASE+X'1000'.
  - R8 points to MYBASE+X'2000'.

It takes extra registers to do it, but that is how to extend the range of a USING past 4K bytes.

Limiting the Range of a USING

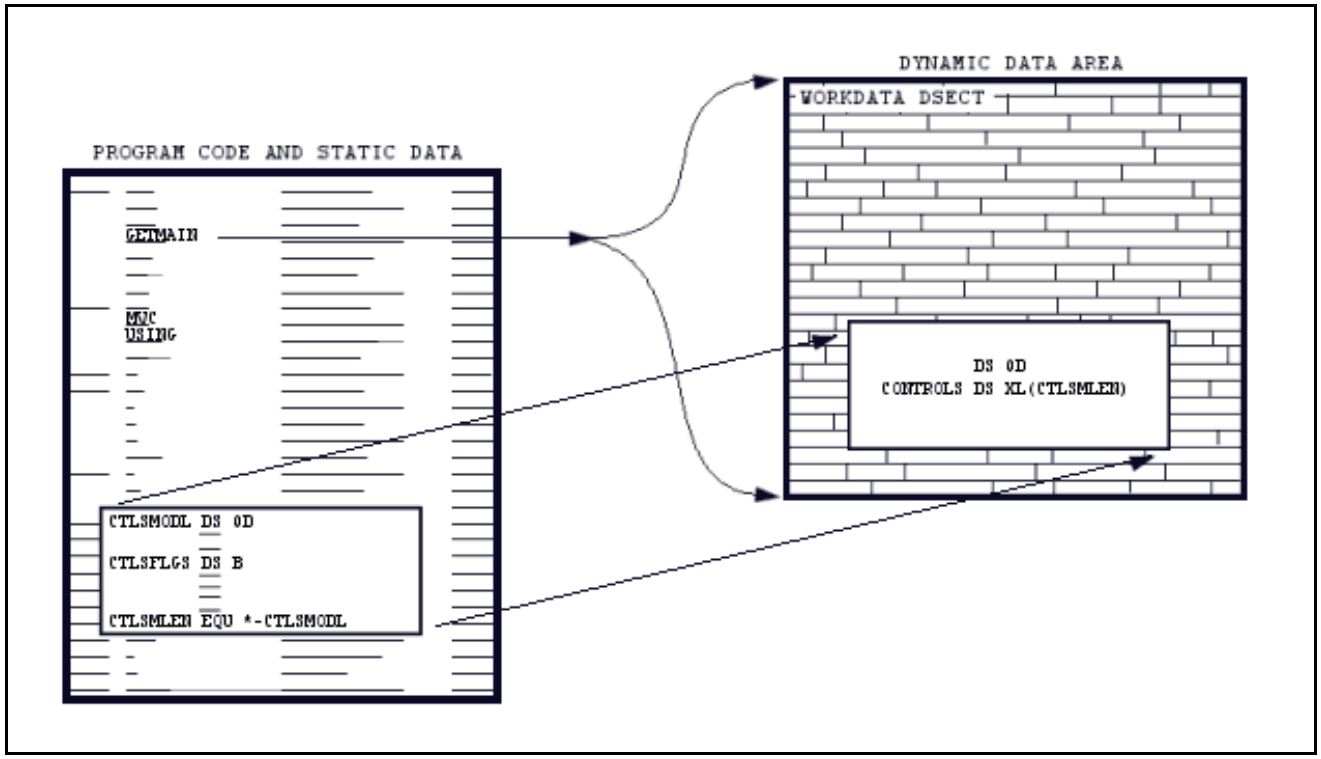
Normally, a USING range is 4096 bytes long, but sometimes it is useful to limit the range of a USING to something less than that. Look at figure **23**. It shows a program that GETMAINs a work area (named WORKDATA in the figure) and then initializes part of that area (the section labelled CONTROLS) with assembled data named

[1] In fact, none are. The CVT's last field is named CVTOSLVF and is located +X'4FF' bytes past CVTTCBP.

[2] The Long Displacement Facility (LDF) is discussed later.

CTLSMODL. Subsequently, the programmer wants to use the those symbols (from the program code's symbol set) that correspond to the CTLSMODL to refer to the copy of that model in the WORKDATA's symbol set.

The mapping of symbols from one set into another can, of course, be done with a dependent USING. To limit that mapping to only those symbols corresponding to a certain range, just specify the starting and ending points of that range, on the USING statement, as a parenthesized pair.



23 Initializing a GETMAIN'd data area from a static model (the diagram)

```
*         ...
          GETMAIN RU,whatever       GET THE WORK AREA
          LR    R8,R1               COPY TO A SAFE REGISTER
          USING WORKDATA,R8         DECLARE ITS BASE
*         ...
          MVC   CONTROLS,CTLSMODL   INITIALIZE
WD        USING (CTLSMODL,CTLSMODL+CTLSMLEN),CONTROLS
*                                   MAP THE MODEL FIELDS (AND *ONLY*
*                                   THE MODEL FIELDS) ONTO THE WORK
*                                   AREA.
*         ...
          TM    WD.CTLSFLGS,whatever
*         ...
          FREEMAIN R,whatever       DONE WITH THE WORK AREA
          DROP  R8                  RELEASE ITS BASE
*         ...
```

24 Initializing a GETMAIN'd data area from a static model (the code)

Figure 24 shows the sequence of code that might be used to accomplish this.

- **GETMAIN RU,whatever**
  LR   R8,R1
  USING WORKDATA,R8
  Once storage for the work area is GETMAIN'd, a USING is issued to make its symbols resolve to the obtained storage.

- **COPYCTLS MVC   CONTROLS,CTLSMODL INITIALIZE**
  WD   USING (CTLSMODL,CTLSMODL+CTLSMLEN),CONTROLS
  Once the CONTROLS buffer within the work area are initialized from CTLSMODL, a dependent USING is issued to map the symbols from the CONTROLS model onto the CONTROLS buffer.
  - The range of the USING is limited to only those symbols that occur within the CONTROLS model.
  - The USING is named so that references to its symbols are clearly understood to be references to the work area instances of those fields, not the model's.

Long Displacement Facility (LDF)

A few years ago, IBM introduced a whole new set of machine instructions that support base-displacement addressing using **20-bit** wide displacement values. (That's 8 bits wider than the old 12-bit fields.) In addition, the wide displacements are **signed**! The consequences of this are:

- For LDF instructions, the range of a USING can now extend **backwards** (as well as forwards) from the base address.

- A single base register can now be used to address an entire **megabyte** of storage, ranging across a distance from **512K prior to** the base address to **512K following** the base address.

But be aware of the limitations:

- This extended addressing is available only for those specific machine instructions that support it (generally, the various 64-bit instructions (LMG, STMY, LG, LGH, ETC.) and several new 32-bit instructions whose names end with the letter Y (LMY, STMY, LY, LHY, etc.).
- These LDF instructions were introduced with IBM's z990 processors, and they are available only on that and newer machines. They are not available on older machines.

Concluding Thoughts

Fundamental concepts to remember:

- USINGs are very powerful and flexible tools for connecting symbol sets to code, to control blocks, and to data in general.
- Symbol sets and program code/data are separate and distinct constructs. Do not get the two confused.
- Code and control blocks are execution time objects. Symbol sets are assembly time objects. Do not get the two timeframes confused.