Chapter 15 - A USING Instruction Tutorial

Introduction

One of the most challenging concepts encountered by students of IBM Assembler Language is that of the USING assembler instruction, or directive. It doesn't generate any object code itself, but it directly influences how machine instructions are assembled. It is a promise by the programmer that certain register(s) will be available at execution time, and so determines which base registers and displacements are chosen by the assembler when converting implicit addresses.

The most common application of USING is in program addressability. The following bit of code is fairly typical (although not necessarily recommended):

```
 1    MAIN     CSECT
 2    STMINST  STM   14,12,12(13)        Save regs; note explicit base (13)
 3             USING MAIN,15             R15 will have A(MAIN) at execution
 4    LAINST   LA    14,SAVEAREA         Get address of new save area
 5             ST    13,4(,14)            and save back pointer in new area
 6             ST    14,8(,13)             and forward pointer in previous
 7             LR    13,14               Set new save area pointer
 8             BALR  12,0                Get address of next inst into R12,
 9             DROP  15                   say we no longer have need of R15,
10             USING PART1,12              and tell assembler our new promise
11    PART1    B     SAVEAREA+18*4       Remainder of program follows
12    SAVEAREA DC    18F'-1'
               ...
```

This bit of code has two USING instructions, one for location MAIN and one for location PART1. The first USING, at statement 3, is a promise that, no matter where the program is loaded at execution time, register 15 can be expected to contain the address of the first instruction of the program. This information is used only during the assembly of the implicit address (of SAVEAREA) in the LA instruction at statement 4, in order to construct the object code of statement 4 with a valid base (R15) and displacement.

The second USING, at statement 10, is a promise that R12 will have the address of PART1 at execution time. Note that this promise will be fulfilled when, at execution time, the BALR instruction at statement 8 is executed.

Two definitions which will be very helpful in describing what follows are the **domain** and the **range** of a USING instruction.

- The **domain** of a USING instruction begins where the USING instruction appears in a program, and it continues until the end of the program, except when:
  - A subsequent DROP instruction specifies the same base register
  - A subsequent USING instruction specifies the same register

- The **range** of a USING instruction is the 4096 bytes within a Control Section (CSECT) beginning at the base address specified in the USING

In order for the assembler to convert implicit addresses to base/displacement, any instruction which refers to an implicit address must be situated within the **domain** of a USING whose **range** includes the referenced location.

Here is the same bit of code with **domain**s and **range**s indicated:

```
                 1    MAIN     CSECT                 <-|
                 2    STMINST  STM   14,12,12(13)      |
Domain of   |->  3             USING MAIN,15           |
USING at    |    4    LAINST   LA    14,SAVEAREA       | Range of USING at
Stmt 3      |    5             ST    13,4(,14)         |   Statement 3 (up
            |    6             ST    14,8(,13)         |    to 4096 bytes)
            |    7             LR    13,14             |
            |    8             BALR  12,0              |
            |->  9             DROP  15                |
Domain of|->    10             USING PART1,12          |
USING at |      11    PART1    B     SAVEAREA+18*4     |  <-| Range of USING
Stmt 10  |      12    SAVEAREA DC    18F'-1'           |    | at Stmt 10 (up

         V                     ...                     V    V to 4096 bytes)
```

In the example above, the **range** of the USING at statement 3 begins at location MAIN and ends at location MAIN+ 4095 (or earlier if the program is shorter than 4096 bytes). This means that location SAVEAREA referenced in statement 4 must be located between MAIN and MAIN+ 4095. The **domain** of that USING begins at statement 3 and ends at statement 9 (the DROP instruction). This means that no instructions following statement 9 can be assembled with R15 as a base register.

Similarly, the **range** of the USING at statement 10 is PART1 through PART1+ 4095. Its **domain** begins at statement 10, so no instruction which preceeds statement 10 can be assembled with R12 as a base register.

Note that the STM instruction at statement 2 is not within the **domain** of any USING, so it cannot have an implicit address for its second operand (which must be given explicitly, as it is in the example). Also, since statement 2 is not within the **range** of the second USING, label STMINST cannot be specified as an implicit address on any instruction following the DROP at statement 9 (assuming only these two USINGs, of course).

It is clearly possible for USINGs to have overlapping **range**s, as is the situation in the example after Statement 10. This is unimportant, though, because the USINGs do not have overlapping **domain**s, and so the assembler has only one USING available for each instruction from which to select a base register. When it happens that there are USINGs with both overlapping **domain**s and overlapping **range**s, the assembler chooses for each implicit operand the base register which will result in the smallest displacement. This fact is important in what follows.

As an aside, it is probably useful to notice that, in general, the instruction which will actually set the promised contents of a base register need not be proximate to the corresponding USING. In our example, the second USING almost directly follows the instruction which will set the register, while the instruction setting the register contents promised by the first USING is nowhere to be seen.

A Closer Look

Now, let's examine what the assembler does when its processes this program, particularly how it handles:

```
 4    LAINST   LA    14,SAVEAREA         Get address of new save area
```

and

```
11    PART1    B     SAVEAREA+18*4       Remainder of program follows
```

Since Statement 4 lies within the **domain** of the USING at Statement 3, and since SAVEAREA is within the **range** of that USING, R15 is an available base register. Since it is the only register available, the assembler chooses it. Since label SAVEAREA is 4+4+4+4+2+2+4=24=X'18' bytes into the USING **range**, the instruction is assembled as **`X'41E0F018'`** [*student: verify this*].

Similarly, since Statement 11 lies within the **domain** of the USING at Statement 10, and since SAVEAREA+18*4 is within the **range** of that USING, R12 is an available base register. Since it is the only base register available (R15 was DROPped), the assembler chooses it. Since SAVEAREA+18*4 is 4+18*4=76=X'4C' bytes into the USING **range**, the instruction is assembled as **`X'47F0C04C'`**.

All of this should be verified by the student. Nothing unusual has been done, and verification should be straightforward. It should be clear that the reason Statement 3 is coded just so is the expectation that R15 ***really will*** have the memory address of the first byte of program code (the STM instruction) at the time of execution. In fact, let's see what would happen if we ***lie*** to the assembler and tell it something which is incorrect! Let's change Statement 3 to pretend that R15 will have the address of LAINST (instead of MAIN) at execution time:

```
 3             USING LAINST,15          R15 will have A(LAINST) at execution
```

Since label SAVEAREA is now 4+4+4+2+2+4=20=X'14' bytes into the USING range, the instruction at Statement 3 is assembled as **`X'41E0F014'`**. Of course, we lied when we wrote the new Statement 3, but the assembler didn't know that and went ahead as though we were telling the truth. Now the object code for statement 4 actually loads the address of four bytes ***before*** SAVEAREA rather than SAVEAREA.

Our conclusion is that the assembler is very gullible! It believes whatever we tell it and then behaves accordingly. It doesn't know what we intend, only what we write. On the one hand, this means we must be careful in writing our USINGs so that the assembler generates the desired object code. On the other hand, this also means that we can trick the assembler into generating the object code we want, in circumstances where telling the truth simply isn't possible. ***The remainder of this note describes how to fool the assembler!*** (Well, not really, of course, but it helps to think of it that way.)

How Can We Use Implicit Addresses Everywhere?

Let's consider that program MAIN above begins a typical first semester exercise for the student of assembler. In this exercise, the student is to "process" two tables. Each table will have, say, up to 25 entries, and each entry will have, say, two fields: name (22 characters) and age (fullword binary), in that order. Before beginning the "process" (whatever that might be), each table entry must be initialized so that the name is blank and the age is zero. Thus, we will have a routine called INIT to which is passed each of the tables, in turn, for initialization: [to simplify the code, the address of the table, rather than a parameter list, is passed]

```
         ...
         LA    1,TABLE1             Get address of first table
         BAL   14,INIT               and pass to INIT
         LA    1,TABLE2             Do the same for the second table
         BAL   14,INIT
         ...
```

When INIT receives control, it does the following:

```
         ...
         SR    0,0                  Initial age value=0
         LA    10,25                Get counter for loop (size of table)
         LR    11,1                 Copy passed address of table
*
LOOP     MVC   0(22,11),BLANKS  *1* Set name to blanks
ST       0,24(,11)              *2*  and age to zero
         LA    11,28(,11)       *3* Get to next entry
*
         BCT   10,LOOP              Continue until done
         ...
BLANKS   DC    CL22' '              Initial value for name
         ...
```

For the purpose of this discussion, let's assume that the data BLANKS is located at X'63A' bytes into the **range** of the USING at statement 10 and that the `LOOP` code lies within the **domain** of that USING. This means that the object code generated by the three statements marked *1*, *2*, and *3* is [*student to verify this*]:

```
D215B000C63A
5000B018
41B0B01C
```

We began with fairly standard source code, except that in the marked statements the explicit form of an operand is used, rather than the implicit form. Explicit operands were used because, at assembly time, there is no way to promise the address to be contained in R11 at execution time. In fact, R11 has a different address each time INIT is called!

This is correct object code, but the explicit addresses create a number of problems:

- There can be a lot of effort involved in calculating offsets and lengths
- The calculations are prone to error
- Most of all, explicit addresses are inflexible; a change to one field may mean that all explicit addresses have to be recalculated

How much nicer it would be to write:

```
         ...
LOOP     MVC   NAME,BLANKS      *1* Set name to blanks
         ST    0,AGE            *2*  and age to zero
         LA    11,NEXTENT       *3* Get to next entry
         ...
```

and get exactly the same object code as with explicit operands! Well, we can, and with all this background, it's actually pretty easy.

The Solution: We Fool the Assembler

Our goal, no matter *how* we accomplish it, is to generate the correct object code. In the case of the instruction at *1*, if we want to use NAME as the first operand, it must be at displacement 0 from base register 11, and it must have a length of 22. For the instruction at *2*, AGE must be on a fullword boundary at displacement 24 from base register 11. And for the instruction at *3*, NEXTENT must be at displacement 28 from base register 11. If we can do this, the same (correct) object code will be generated.

The solution has two parts. First, we define fields labeled as we want, in the format we want. Let's place them just after BLANKS:

```
         ...
         DS    0F                    Assure correct boundary alignments
BLANKS   DC    CL22' '               Initial value for name
NAME     DS    CL22                  Dummy NAME field for tables
AGE      DS    F                     Dummy AGE field for tables
NEXTENT  DS    0H                    Dummy next entry in a table
         ...
```

Now we trick the assembler. Since register 11 is the one we want as our base register, we tell the assembler that at execution time R11 will have the address of NAME. This is a lie, of course, because R11 will actually point to a table entry; but the assembler will believe us and will generate the correct object code. Here's what we do: adding one USING statement (and one DROP) to our original LOOP code, we place our marked instructions in a new USING **domain** (which overlaps the program USING **domain** established at Statement 10).

```
         ...
         SR    0,0                    Initial age value=0
         LA    10,25                  Get counter for loop (size of table)
         LR    11,1                   Copy passed address of table to R11
*                                      then lie to the assembler!
         USING NAME,11                Begin the domain of the lie
LOOP     MVC   NAME,BLANKS        *1* Set name to blanks
         ST    0,AGE              *2*  and age to zero
         LA    11,NEXTENT         *3* Get to next entry
         DROP  11                     End the domain of the lie
*
         BCT   10,LOOP                Continue until done
         ...
         DS    0F                     Assure correct boundary alignments
BLANKS   DC    CL22' '                Initial value for name
NAME     DS    CL22                   Dummy NAME field for tables
AGE      DS    F                      Dummy AGE field for tables
NEXTENT  DS    0H                     Dummy next entry in a table
         ...
```

We must be careful, though; since the marked instructions lie within two overlapping USING **domain**s and since they refer to operands which lie within two overlapping USING **range**s (the new one begins at NAME and continues through NAME+ 4095), the assembler can choose between base registers 12 and 11 for each implicit address which lies within the overlapping **range**s.

For the instruction marked *1*, the first operand, NAME, is at offset X'650' in the **range** of the USING at Stmt 10 (remember that BLANKS is at X'63A'), while it is at offset X'000' in the **range** of the USING just before LOOP. Therefore, following the rule mentioned at the end of the Introduction, the assembler chooses to use the base register which gives the smallest displacement, and the NAME operand is assembled with X'B000' as base/displacement (rather than X'C650'). Since the definition of BLANKS lies *outside* the **range** of the new USING, the assembler can choose only R12 and assembles the BLANKS operand of *1* as X'C63A'. Since this is the desired object code, the lie has worked! *The student should verify that *2* and *3* also assemble correctly.*

The student should also notice that at no time are the storage areas defined by NAME, AGE, and NEXTENT actually referenced! Base register R11 always has the address of a table entry and never has the address of NAME. This fact will shortly help us take the last step in solving our problem.

If it weren't for the potential problem introduced by overlapping **range**s, the solution would be nearly complete. Unfortunately, overlapping **range**s make it easy for errors to be introduced. Consider what would happen if our storage areas were defined with BLANKS *after*, rather than before NAME:

```
         ...
         DS    0F                     Assure correct boundary alignments
NAME     DS    CL22                   Dummy NAME field for tables
AGE      DS    F                      Dummy AGE field for tables
NEXTENT  DS    0H                     Dummy next entry in a table
BLANKS   DC    CL22' '                Initial value for name
         ...
```

Now BLANKS lies within overlapping USING **range**s and the assembler, following the smallest displacement rule, will choose to assemble the statement marked *1* as:

```
D215B000B01C
```

which is clearly incorrect object code [*student: explain why this happened*]. It appears that our lie has gotten us into trouble and we need to find a way out.

The Last Step: We Eliminate Overlapping USING Ranges

Our best bet is to find a way to define the labels we want (NAME, AGE, and NEXTENT) so that they are outside any other possible USING **range**. That way, the assembler will choose R11 as the base register *only* when implicitly addressing those labels. Since a USING **range** is 4096 bytes, we could make the program longer than 4096 bytes by inserting some large storage areas, then placing our fields after them, perhaps just before END.

There is a better answer, though. It turns out that USING **range**s are limited to addresses within a single control section, so our last step will be to define our labels in another control section. Here is one possibility:

```
         ...
BLANKS   DC    CL22' '               Initial value for name
         ...
ENTRY    CSECT
NAME     DS    CL22                  Dummy NAME field for tables
AGE      DS    F                     Dummy AGE field for tables
NEXTENT  DS    0H                    Dummy next entry in a table
*
         END   MAIN
```

and here is another:

```
MAIN     CSECT ,                     Beginning of program
         ...
*
ENTRY    CSECT
NAME     DS    CL22                  Dummy NAME field for tables
AGE      DS    F                     Dummy AGE field for tables
NEXTENT  DS    0H                    Dummy next entry in a table
*
MAIN     CSECT ,                     Resume original control section
BLANKS   DC    CL22' '               Initial value for name
     ...
```

In both cases, the labels NAME, AGE, and NEXTENT now lie outside the range of the USING at Statement 10 because they are located within a different control section. The assembler is forced to choose R12 as the base register for BLANKS in *1*, and will correctly choose R11 as the base register for NAME. That' s it - we now generate the correct code in all circumstances because we eliminated overlapping USING **range**s by isolating our labels in a different control section.

We can extend this solution to deal with situations where we want to use this trickery in multiple places within our program. Although we avoided parameter lists earlier for simplicity, they are the normal method to pass information to a called routine. Thus we might call INIT with the following sequence:

```
         ...
         LA    1,=A(TABLE1)
         BAL   14,INIT
         LA    1,=A(TABLE2)
         BAL   14,INIT
         ...
```

We now add an INITPRM CSECT (defining ATABLE) and rewrite INIT as:

```
MAIN     CSECT ,                     Beginning of program
         ...
INIT     DS    0H
         ...
         SR    0,0                   Initial age value=0
         LA    10,25                 Get counter for loop (size of table)
         USING ATABLE,1              Trick assembler to gen correct code
         L     11,ATABLE             Copy passed address of table to R11
         DROP  1                      then end USING domain for R1
*
         USING NAME,11               Begin the domain of the lie
LOOP     MVC   NAME,BLANKS       *1* Set name to blanks
         ST    0,AGE             *2*  and age to zero
         LA    11,NEXTENT        *3* Get to next entry
         DROP  11                    End the domain of the lie
*
         BCT   10,LOOP               Continue until done
         ...
*
ENTRY    CSECT
NAME     DS    CL22                  Dummy NAME field for tables
AGE      DS    F                     Dummy AGE field for tables
NEXTENT  DS    0H                    Dummy next entry in a table
*
INITPRM  CSECT
ATABLE   DS    A                     Address of passed table
*
MAIN     CSECT ,                     Resume original control section
BLANKS   DC    CL22' '               Initial value for name
         ...
```

Now we are done! We've found a way to fool the assembler into generating the object code we need in circumstances where it isn't possible to get the result "truthfully." By eliminating overlapping USING ranges, we've also assured that incorrect base registers are less likely to be chosen. By isolating each group of storage areas associated with a "trick" in a separate control section, we have reduced potential errors which might arise from overlapping USING ranges in a "multiple trick" situation. There is only one minor, helpful but unnecessary change left which we can make if we wish.

The Final Last Step: We Stop Wasting Space

We mentioned earlier that none of the *storage areas* created for the purpose of tricking the assembler is ever actually referenced or changed. This is the result of our trick, and it leaves these storage areas doing nothing but wasting space. All they are needed for is to make it possible for the assembler to see the offsets and lengths of the various labels.

Since the storage isn't needed, just the labels, we can change the control sections to be *dummy* control sections. Dummy control sections reserve no storage; they are simply sets of labels for the assembler to refer to in situations such as we have created here for our trick. The savings in storage is minimal in our example, but in "real" programs the reduction can be noticeable.

Dummy control sections are implemented by using the DSECT instruction in place of CSECT. This would mean that our labels would be defined as follows:

```
         ...
*
ENTRY    DSECT
NAME     DS    CL22                    Dummy NAME field for tables
AGE      DS    F                       Dummy AGE field for tables
NEXTENT  DS    0H                      Dummy next entry in a table
*
INITPRM  DSECT
ATABLE   DS    A                       Address of passed table
*
MAIN     CSECT ,                       Resume original control section
BLANKS   DC    CL22' '                 Initial value for name
         ...
```

Thus, while MAIN is a CSECT, ENTRY and INITPRM are DSECTs which take up no storage.

Conclusion

Assembler programming can be greatly simplified by means of the USING instruction. With the sleight-of-hand described in this note, the assembler can be convinced to generate correct object code from implicit addresses in situations where implicit addresses might not seem possible. By isolating labels used in the trickery into CSECTs or DSECTs, potential addressing errors are reduced. By use of DSECTs, wasted memory is reduced.

MS, March 1998