Chapter 10: Assembler For Dummies

In which we learn that working with dummies can be the smart thing to do.

We begin this chapter by reviewing two assembler directives.

USING and DROP

The assembler converts many of the symbols we use into base/displacement addresses. Each address contains a base register and a displacement from a known address in our program. In building a program, we have control over which registers and base addresses are used at various points in the program. A USING statement is a directive to the assembler that controls which registers will be selected and the locations of the base addresses of those registers.

An ordinary USING statement is typically coded by naming a base address (most often a relocatable expression) in operand 1, followed by one or more registers. If * is used as a base address, it refers to the current value of the location counter. A base address is associated with each named register. The base address specified in operand 1 is associated with the first named register. If multiple registers are coded, the base address in increased by 4k for each subsequent register. Here are some examples,

```
USING    *,R12           BASE ADDRESS FOR R12 = CURRENT LOCATION
                         BASE REGISTER = 12

USING    *,R12,R11,R10   BASE ADDRESS FOR R12 = CURRENT LOCATION
                         BASE ADDRESS FOR R11 = CURRENT LOCATION+4096
                         BASE ADDRESS FOR R10 = CURRENT LOCATION+8192
                         BASE REGISTERS = 12,11,10

USING    HERE,R12,R11    BASE ADDRESS FOR R12 = ADDRESS OF HERE
                         BASE ADDRESS FOR R11 = ADDRESS OF HERE+4096
                         BASE REGISTERs = 12,11
```

It's a common practice for programmers to use 12 as the first base register and work backwards in sequence if more base registers are needed. This is not a requirement, but simply a programming convention. The use of register 0 in a USING statement as a base register is a rare occurrence and beyond the scope of this discussion.

In processing a USING statement, the assembler records each register and its associated base address in a table called the ***USING table.*** The assembler consults the USING table whenever it converts a symbolic name to a base/displacement address. Registers are first added to the table by coding them in a USING statement. The base address associated with a register can be changed in the table by recoding the same register in a USING with a different base address. Finally, a register can be removed from the USING table altogether by coding it in a DROP statement. Here are some typical DROP statements,

```
DROP   R12
DROP   R12,R11,R10
```

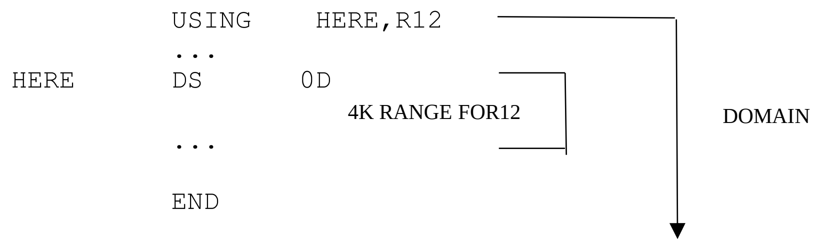
As programs grow larger we often need to add more base registers, and as a result, available registers can become scarce. The DROP statement gives us the ability to tell the assembler to stop using a specific register for base/displacement addressing. This frees the register for other uses. By dropping a register, you also prevent the assembler from choosing a register inadvertently for a base/displacement address in an area of the program where you didn't intend for it to be selected.

There are two important definitions associated with USING and DROP.

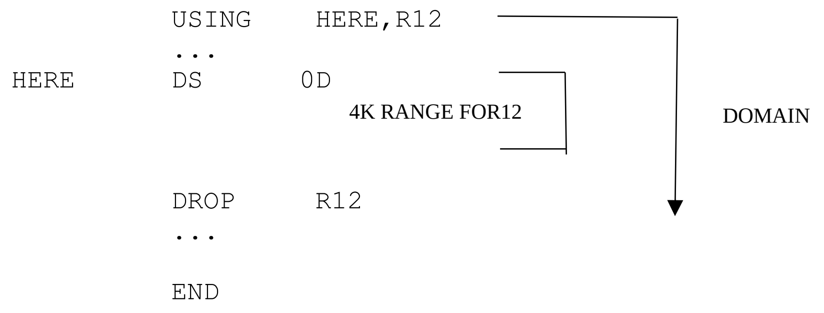
1) A USING **domain** starts at the USING statement and continues to the end of the program, or until the point at which all the registers named in the USING have been dropped.

2) A USING **range** starts at the base address and continues for 4096 bytes (4k) of storage.

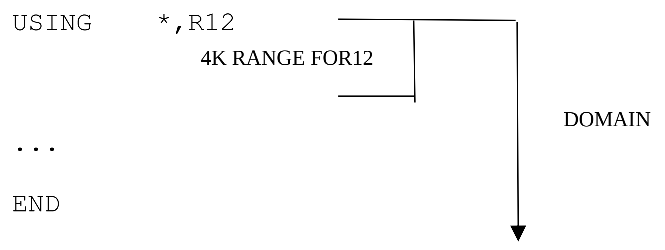
Here are several examples:



```
          USING    HERE,R12
          ...
HERE      DS       0D

          ...

          END
```

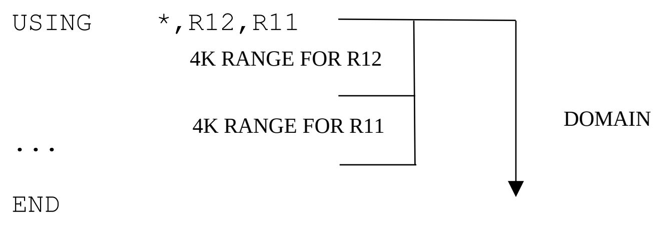


```
          USING    HERE,R12
          ...
HERE      DS       0D


          DROP     R12
          ...

          END
```



```
          USING    *,R12



          ...

          END
```

If a USING statement names several registers, consecutive 4k ranges are created for each register.



```
USING    *,R12,R11

...

END
```

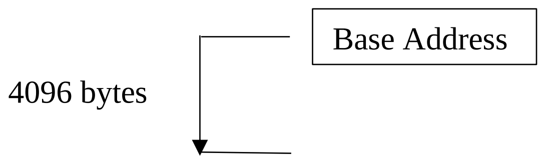
Addressability Rules

In order for the assembler to generate base/displacement addresses for symbols (and avoid addressability errors), the code you write has to conform to two rules concerning domains and ranges.

1) The **definition** of each symbol must occur within the **range** of some USING statement. (Otherwise, the assembler can't generate a displacement for the symbol.)

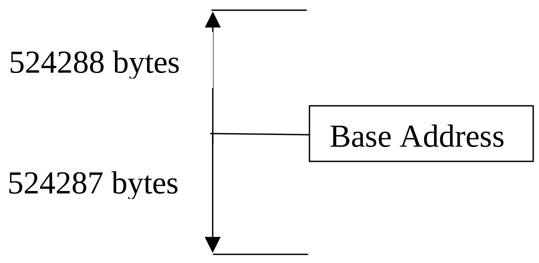
2) The **use** of a symbol in an instruction must occur with the **domain** of the corresponding USING statement. (Otherwise, the assembler can't select a base register for the symbol.)

Let's give some further attention to rule 1, and what it means to be within the range of a USING statement. In the original architecture, instructions used 12-bit displacements which were interpreted as plain binary values. This provided displacements from 0 to 4095. The symbols that could be addressed by these instructions had to be defined within a 4k range starting at the base address and moving forward. These same instructions are still heavily used today.

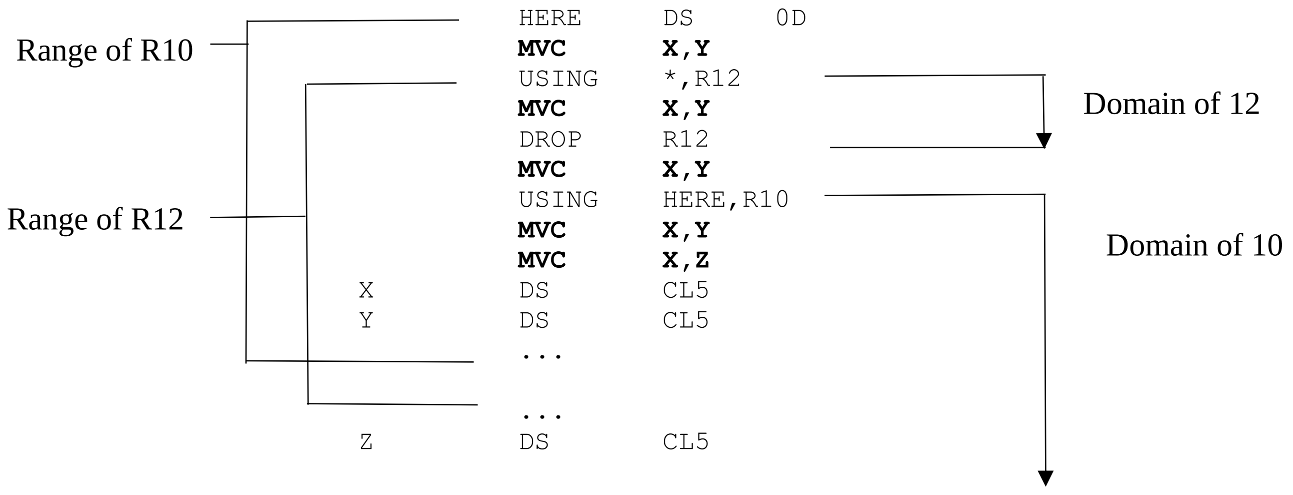


But modern machines also support some instructions that use 20-bit displacements. Furthermore, the twenty bits are interpreted as twos-complement integers, which means they can be positive or negative. The range for these instructions is forward from the base address by up to 524287 bytes or backward from the base address by 524288 bytes. A 20-bit signed displacement satisfies the inequality

$$-524288 \leq \text{displacement} \leq +524287$$



Consider the following example code and the addressability errors the MVCs generate. The code is an experiment in assembling and is not designed to be executed. In order to get the code to be executable, we would also have to load the base registers properly at runtime.



```
            HERE      DS      0D
            MVC       X,Y
            USING     *,R12
            MVC       X,Y
            DROP      R12
            MVC       X,Y
            USING     HERE,R10
            MVC       X,Y
            MVC       X,Z
X           DS        CL5
Y           DS        CL5
            ...

            ...
Z           DS        CL5
```

- The first MVC generates an addressability error because the **use** of X and Y are not in the domain of any USING statement. The assembler can't select a base register.

- The second MVC assembles properly. The **use** of X and Y occurs in the domain of R12, and the **definitions** of X and Y occur in the corresponding 4k range of 12. The base/displacement addresses are C018 and C01D. (This is easier to see in the listing which follows.) C corresponds to register 12, and 018 and 01D are the displacements to X and Y from the base address of register 12.

- The third MVC generates an addressability error because we dropped R12 and at this point there are no available base registers in the base register table.

- The fourth MVC assembles properly. The use of X and Y occurs in the domain of R10, and the definitions of X and Y occur in a 4k range starting at the corresponding base address, HERE. The base/displacement addresses are A01E and A023. The *A* corresponds to register 10, and 01E and 023 are the displacements to X and Y from base address HERE.

- The fifth MVC generates an addressability error because Z is **defined** outside all ranges.

There is an important lesson we can learn here: The executable code we write can look correct if we aren't paying attention to the USING statements we have written. It's possible to write executable code that is correct but assembles incorrectly because of our USING and DROP directives. The solution to this problem is to learn to read the object code that appears on the left of each instruction in our assembly listing. For some subtle problems, this is a necessity.

Here is an assembly listing of the previous code. I've highlighted the assembled code for the the two MVCs that assembled properly. The executable code is identical, but the object code uses different base registers. Can you see why?

```
000000                                2 HERE     DS   0D
000000 0000 0000 0000 00000 00000     3          MVC   X,Y
** ASMA307E No active USING for operand X
** ASMA307E No active USING for operand Y
** ASMA435I Record 3 in KC02486.KC02486B.JOB09826.D0000101.? on volume:
                 R:C  00006           4          USING *,R12
000006 D204 C018 C01D 0001E 00023     5          MVC   X,Y
                                      6          DROP  R12
00000C 0000 0000 0000 00000 00000     7          MVC   X,Y
** ASMA307E No active USING for operand X
** ASMA307E No active USING for operand Y
** ASMA435I Record 7 in KC02486.KC02486B.JOB09826.D0000101.? on volume:
                 R:A  00000           8          USING HERE,R10
000012 D204 A01E A023 0001E 00023     9          MVC   X,Y
000018 0000 0000 0000 00000 00000    10          MVC   X,Z
** ASMA034E Operand Z beyond active USING range by 945 bytes
** ASMA435I Record 10 in KC02486.KC02486B.JOB09826.D0000101.? on volume:
00001E                               11 X        DS    CL5
000023                               12 Y        DS    CL5
000028                               13          DS    CL5000
0013B0                               14 Z        DS    CL5
```

As a practical matter, when you get addressability errors, they often occur in bunches. The good news is that fixing one addressability error can repair numerous others. Keep in mind that there are only two causes for all addressability errors. Either the assembler can't choose a base register for your problem symbol, or the assembler can't generate a displacement to your symbol. If you have addressability errors, start with the first error in your code and try to repair it. You may find that fixing the first error fixes many others.

How the Assembler Employs the USING Table

If you code this,

```
        MVC   X,Y
```

the assembler has to convert X and Y to base/displacement addresses. How does this occur? First of all, IBM's assembler is a two-pass assembler. This means the assembler reads your source code twice.

- **Pass One** - The first time the code is read, the goal is to build a symbol table that contains each symbol you defined (the names starting in column 1), and the location counter value associated with each symbol. In other words, the assembler is trying to determine where the important parts of your program are located. The symbols and their locations are recorded in the symbol table which is used in pass two.

- **Pass Two** – After the symbol table is completed in pass one, the assembler knows the relative location of every symbol in your program. The goal in pass two is to generate code for each instruction and all the storage areas that need to be initialized. As the assembler reads through the code, it dynamically builds the USING table (adding or dropping registers) so that it knows which registers are available at any point for converting symbols to base/displacement addresses.

When the assembler is processing the MVC instruction above, it can consult the symbol table built in pass one for the location of X. Let's assume that X has location x'00624'. For purposes of discussion, let's assume the USING table looks like this at the time the assembler is assembling the instruction.

Register	Base Address
12	x'00200'
11	x'00500'
10	x'00700'

The assembler searches the table sequentially looking for available registers. Which registers will work? Register 12 is associated with address x'00200', so a base/displacement address for X that uses 12 will need a displacement of x'00624' – x'00200' = x'00424'. The base/displacement address for X would become C424. Register 11 is associated with address x'00500', so a base/displacement address for X that uses 11 will need a displacement of x'00624' – x'00500' = x'00124'. The base/displacement address for X would become B124. Register 10 can't be used to form an address for X since the base address for 10, x'00700', occurs after the location of X, x'00624', and the MVC instruction requires a 12-bit non-negative displacement.

So, after searching the table, the assembler finds there are two registers that can be used to create a base/displacement address. When this happens, the assembler chooses the register that generates the smaller displacement. In this case, the base/displacement address of X would become B124. In the rare case where two registers generate the same displacement for a symbol's address, the assembler will choose the larger-numbered register.

Suppose the location of Y in the symbol table were x'00918'? What base/displacement would the assembler generate for Y? If you concluded it would be A218, you are well on your way to understanding DROPs and USINGs. Now that we have reviewed DROP and USING, we have a better idea of how the assembler handles base/displacement addresses. It's time to move on to the main topic of this chapter.

DSECTs – Dummy Sections

A DSECT or Dummy Section provides a programmer with the ability to describe the layout of a storage area without reserving virtual storage for it. The DSECT layout can then be used to reference any storage area which is addressable by the program. Think of the DSECT as a template or pattern, which can be associated with an area of storage. Once the DSECT is "positioned", the symbolic names in the DSECT can be used to extract or change data in the underlying storage area. DSECTs provide the ability to assign symbols to storage areas dynamically – a powerful tool.

Let's start by creating a DSECT called CUSTOMER that describes three storage fields.

```
CUSTOMER  DSECT
FNAME     DS     CL10
LNAME     DS     CL10
BALANCE   DS     PL5
```

The CUSTOMER DSECT is created by coding the DSECT directive, which indicates the beginning of the dummy section. The general format for this directive is listed below.

```
NAME     DSECT   COMMENT
```

The pattern for the DSECT is created using a series of DS directives to describe a collection of fields in storage: FNAME, LNAME, and BALANCE. You can also use DC's inside a DSECT, but remember that a DSECT acquires addressability dynamically. There is no storage area automatically associated with the DSECT. As a result, if you code a DC inside a DSECT, **no storage is initialized**. The assembler will consider the length of the constant in any calculations, but the constant is not created in memory. You should also keep in mind that the length associated with the DSECT name is 1 (L'CUSTOMER = 1). If you need the length of all the fields in the DSECT, you have to create a symbol inside the DSECT that references everything. For example, CUST references the entire DSECT below.

```
CUSTOMER  DSECT
CUST      DS    0CL25
FNAME     DS     CL10
LNAME     DS     CL10
BALANCE   DS     PL5
```

The end of a DSECT is indicated by the beginning of a CSECT or another DSECT.

DSECT Addressability

After creating a DSECT, we establish addressability for its fields in a two-step process:

1) A register is associated with the DSECT name by coding a USING directive that names the DSECT and a register,

2) The associated register is loaded with the address of a storage area to be addressed.

In this way, the DSECT can be used to access any available storage we can properly address. Let's use this idea to extract the first two customer entries, CUST1 and CUST2, below. Notice the subfields of these group items do not have symbolic names.

```
MAIN     CSECT
         ...
         USING CUSTOMER,R7    ASSOCIATE R7 WITH CUSTOMER DSECT (STEP 1)
* R7 NOW CONTROLS THE DSECT
         LA   R7,CUST1        ASSOCIATE THE DSECT WITH CUST1 (STEP 2)
         MVC  NAME1,FNAME     FNAME REFERENCES FRED
         MVC  NAME2,LNAME     LNAME REFERENCES SMITH
         ZAP  BALOUT,BALANCE  BALANCE REFERENCES 432.98
* GO AFTER CUST2
         LA   R7,CUST2        ASSOCIATE THE DSECT WITH CUST2 (STEP 2)
         MVC  NAME1,FNAME     FNAME REFERENCES SUSAN
         MVC  NAME2,LNAME     LNAME REFERENCES JONES
         ZAP  BALOUT,BALANCE  BALANCE REFERENCES 123.45
         DROP R7              END ADDRESSABILITY FOR R7
         ...
* FIRST CUSTOMER
CUST1    DS  0CL35
         DC   CL10'FRED'
         DC   CL20'SMITH'
         DC   PL5'432.98'

* SECOND CUSTOMER
CUST2    DS  0CL35
         DC   CL10'SUSAN'
         DC   CL20'JONES'
         DC   PL5'123.45'
         ...
NAME1    DS   CL10
NAME2    DS   CL10
BALOUT   DS   PL5
```

After defining the dummy section above, we associate it with an available register in a USING directive. In the example code, the statement `USING CUSTOMER,R7` associates register seven with the `CUSTOMER` dummy section. Addressability is established when the register is loaded with the address of a `CUST1` using the following line,

```
LA   R7,CUST1
```

At that point, `FNAME` contains the value “`FRED      `”, `LNAME` contains the value “`SMITH     `”, and `BALANCE` contains a packed decimal 432.98. Loading the register has “positioned” the DSECT on the storage area whose address is contained in the register, and has given us access to the subfields of `CUST1`. We repeat the technique by loading R7 with the address of `CUST2`.

```
LA   R7,CUST2
```

After the second load address, `FNAME` contains the value “`SUSAN     `”, `LNAME` contains “`JONES     `”, and `BALANCE` contains a packed decimal 123.45.

When we have finished using the fields in the DSECT, we issue a `DROP` directive that instructs the assembler to stop using register 7 with the associated DSECT. By coding a `DROP` for `R7`, we insure that register 7 will not be inadvertently chosen as a base register later in a different context.

Consider the power of the above technique. There were no symbolic names associated with the subfields of the first or second customers above. By creating a DSECT, we have dynamically assigned symbols to storage areas and we have accessed those data areas. DSECTs have many uses in assembly language, including array processing, locate mode I/O, and data structure creation and manipulation. Most sophisticated assembler programs use DSECTS. Next, let’s consider the use of DSECTs in array processing as a way of continuing our study.

Loading an Array with Data

In the following example, we load a storage area with data that is read from a file. The first 100 records in the file will be read and stored in a table. We assume the file has at least 100 records. The FILEIN DCB is not specified, and we will use a DSECT similar to the previous example.

```
CUSTOMER DSECT
FNAME    DS     CL10
LNAME    DS     CL10
BALANCE  DS     PL5
CRECLEN  EQU    *-CUSTOMER       LENGTH OF CUSTOMER ENTRY
```

```
MAIN CSECT
         ...
         USING CUSTOMER,R7       R7 CONTROLS DSECT POSITION
         LA    R7,TABLE          POSITION DSECT AT TOP OF TABLE
         L     R4,NOITEMSF       LOAD # OF TABLE ENTRIES INTO R4
LOOP     EQU *
         GET   FILEIN,RECIN      READ A RECORD
         MVC   FNAME,FNAMEIN     MOVE DATA ...
         MVC   LNAME,LNAMEIN     ... FROM RECIN ...
         ZAP   BALANCE,BALIN     ... TO THE TABLE
         LA    R7,CRECLEN(R0,R7) BUMP THE DSECT TO NEXT TABLE ITEM
         BCT   R4,LOOP           BRANCH BACK IF MORE RECS
         ...
NOITEMS  EQU   100               TABLE WILL HAVE ROOM FOR 100 ITEMS
NOITEMSF DC    A(NOITEMS)        NO OF ITEMS AS A FULLWORD
TABLE    DS    (NOITEMS)CL(CRECLEN)  AREA FOR 100 CUSTOMER RECORDS

RECIN    DS   0CL80
FNAMEIN  DS    CL10
LNAMEIN  DS    CL10
BALIN    DS    PL5
         DS    CL55
```

The table's address is placed in register 7 by the first `LA` instruction.

```
LA    R7,TABLE
```

This has the effect of positioning the DSECT at the beginning of the table area since register 7 is associated with `CUSTOMER` in the `USING` statement. A record is read, and the subfields are moved into the table by two `MVCs` and a `ZAP`. The most interesting line of code occurs after we have processed the first customer entry,

```
 LA  R7,CRECLEN(R0,R7)
```

Here we are loading an address into register 7. The effective address of Operand 2 is computed by adding `CRECLEN`, a pure integer representing the length of a single customer entry, and the contents of register 7. (`R0` indicates that the index register is ignored in the calculation.) This adds 25 to the contents of register 7 and puts the sum back into register 7, effectively "bumping" the DSECT forward 25 bytes. We loop back, read another record, and move it to the next area in the table. We continue in this way until 100 records are loaded. Register 4 is used to control the loop.

To move the DSECT to the next empty table entry, we coded the following line,

```
LA R7,CRECLEN(R0,R7) BUMP THE DSECT
```

This type of statement is often used in DSECT processing, so we consider it in some detail. Even though the statement uses explicit addressing in operand 2 - a technique we usually avoid - the technique is regarded as an acceptable practice here. The explicit format for operand 2 is `D2(X2,B2)` where `D2` is the displacement, `X2` is an index register, and `B2` is a base register. The effect of the instruction is to compute the second operand address explicitly from the base register (`R7`), index register (`R0`), and displacement (`CRECLEN`). When 0 is specified as a register, that component of the address computation is ignored. The "effective" address consists of the contents of register 7 plus the displacement of 25. Since register 7 contains an address of a table entry, adding 25 to the register will create the address of the next table entry. The DSECT has been "bumped" down to the next table entry. Finally, the BCT instruction is used for loop control and ensures that each entry in the table is filled.

The LA instruction above can also be acceptably coded like this,

```
LA R7,CRECLEN(,R7)
```

The comma indicates that the index register should default to 0 and does not participate in the address calculation. There are two other formats for LA that will work, but should be avoided,

```
LA R7,CRECLEN(R7)
LA R7,CRECLEN(R7,)
```

Both of these lines use to 0 by default as a base register. This is considered bad form.

Where Do DSECTS Live?

The assembler provides great freedom in the placement of DSECTs. If you are just beginning to experiment with DSECTs, you might start by placing your DSECTs above the main CSECT like those in the example below:

```
CUSTA    DSECT
A        DS    CL3
B        DS    CL4
CUSTB    DSECT
C        DS    CL4
D        DS    CL10
MAIN     CSECT
         ...
         END   MAIN
```

The CUSTA DSECT ends when the CUSTB DSECT begins. The CUSTB DSECT ends when the MAIN CSECT begins. The MAIN CSECT ends with the END statement.

Another option is to put your DSECTS at the end like this:

```
MAIN      CSECT
           ...
CUSTA    DSECT
A        DS    CL3
B        DS    CL4
C        DS    CL10
CUSTB    DSECT
D        DS    CL4
E        DS    CL10
MAIN     CSECT
         ...
         END   MAIN
```

After coding the two DSECTS, we must restart the main control section before coding END.

Like CSECTs, DSECTs can also be stopped and started again. The assembler will put the correct pieces together in the right order, so the following example is equivalent to the example above.

```
MAIN     CSECT
          ...
CUSTA    DSECT
A        DS     CL3
B        DS     CL4
CUSTB    DSECT
D        DS     CL4
CUSTA    DSECT
C        DS     CL10
CUSTB    DSECT
E        DS     CL10
MAIN     CSECT
          ...
         END    MAIN
```

The `CUSTA` DSECT consists of three fields: `A`, `B`, and `C`. The `CUSTB` DSECT consists of two fields: `D` and `E`. The idea of mixing CSECTs and DSECTS is a bit advanced for a beginning text, but you should be aware that it's possible.

The Magic of DSECTs

How does this DSECT magic work? When the assembler processes a DSECT, it resets the location counter for the DSECT to 0 to compute displacements to fields inside.

The example below lists a DSECT and the corresponding location counter values.

```
LOC
            CUSTOMER DSECT
000000      FNAME    DS      CL10
00000A      LNAME    DS      CL10
000014      BALANCE  DS      PL5
```

If we code "`USING  CUSTOMER,R7`", the assembler will use 7 as the base register along with the appropriate displacement above when creating base/displacement addresses. For example, the base/displacement address for `BALANCE` will be `7014`. Everything will work out as long as register 7 contains the appropriate runtime address. When you are using DSECTs, the USING statement's associated register controls the positioning of the DSECT. This is backward from the way normal symbols work, where we fix a base address in the register and vary the displacement to get to the field we are addressing. With DSECTs, the offsets are fixed, and we vary the contents of the base register. This has an important implication: If a register contains the address of something you want, you can use a DSECT to provide a symbolic reference to it. Of course, it has to be something in your region, otherwise you will get a S0C4.

Labeled USING Statements

Typically, with an ordinary USING statement, you associate the name of a DSECT with a single register. The addressability of symbols in the DSECT are controlled by the address in the associated register. But it's often the case that you would like to use a DSECT and the symbols it defines with two different memory locations at the same time. For example, you might be reading and writing records that have the same structure. Or you might be processing a linked list where you need to access two items in the list that have identical structures at the same time. With ordinary USING statements, the only way to do that is to create two different DSECTs that have similar structures and associate them with two different registers. This is not an ideal solution.

Labeled USING statements were designed to address this problem by adding a label in front of the USING that can be used to qualify the DSECT symbols. Here is the format of a simple labeled USING statement,

label USING *base address*, *base register*, …

To illustrate how labeled USINGs work, let's reuse the CUSTOMER DSECT,

```
CUSTOMER DSECT
FNAME    DS     CL10
LNAME    DS     CL10
BALANCE  DS     PL5
CRECLEN  EQU    *-CUSTOMER        LENGTH OF CUSTOMER ENTRY
```

We again assume the definitions of two customers,

```
* FIRST CUSTOMER
CUST1    DS  0CL35
         DC   CL10'FRED'
         DC   CL20'SMITH'
         DC   PL5'432.98'

* SECOND CUSTOMER
CUST2    DS  0CL35
         DC   CL10'SUSAN'
         DC   CL20'JONES'
         DC   PL5'123.45'
```

To access both customers, we place labels on two USING statements referencing the same DSECT and load each register with the address of a customer.

```
FIRST    USING  CUSTOMER,R6
SECOND   USING  CUSTOMER,R7
         LA     R6,CUST1
         LA     R7,CUST2
```

With the registers loaded, we have addressability to both customers. We can disambiguate the common symbols by qualifying symbols with the appropriate label, and copy the data from the first customer to the second customer.

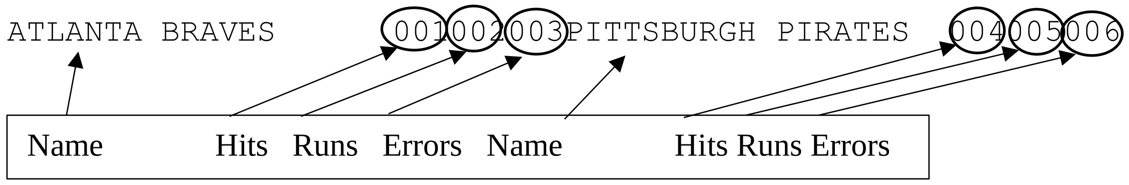
```
         MVC    SECOND.FNAME,FIRST.FNAME
         MVC    SECOND.LNAME,FIRST.LNAME
         ZAP    SECOND.BALANCE,FIRST.BALANCE
```

Dropping a register that is named in a labeled USING requires special handling. To end the domain of a labeled USING instruction, you must code a DROP instruction with an operand that specifies the label of the labeled USING instruction.

```
               DROP   FIRST
               DROP   SECOND
```

Dependent USING Statements

Occasionally, we need to process a data structure that contains a substructure that has its own DSECT. As a simple illustration, consider reading a collection of baseball records where each record describes a single game. Here is a typical record,



The statistics for each team consist of hits, runs, and errors, so we build a DSECT to describe those items.

```
      STATS    DSECT
      RUNS     DS    CL3
      HITS     DS    CL3
      ERRORS   DS    CL3
      STATSLEN EQU   *-STATS
```

We then use that DSECT to build a record structure that describes a game with each team’s name and statistics.

```
      GAME     DSECT
      TEAM1    DS    CL20
      GAMESTA1 DS    CL(STATSLEN)
      TEAM2    DS    CL20
      GAMESTA2 DS    CL(STATSLEN)
      GAMELEN  EQU   *-GAME
```

We can then use an ordinary USING statement to establish addressability for the `GAME DSECT` to an input record buffer called `RECIN`,

```
               USING  GAME,R5
               LA     R5,RECIN
```

How do we get addressability to the statistics for the first team? We could use another base register to establish addressability for the `STATS DSECT`, but that seems wasteful, since we already have addressability to every field in the `GAME DSECT` through register 5. An alternative to using another register is to code a Dependent USING statement that names `STATS` as the DSECT, and a relocatable symbol (to which we already have addressability) as the base. Here is an example,

```
                  USING  STATS,GAMESTA1
```

Addressability for `STATS` is dependent on addressability to `GAMESTA1`. Register 5 will be used as the base register for all the fields in `STATS`. We have saved a register!

How would we address the fields in `GAMESTA2`? One problem with coding dependent USING statements involves dropping the domain. The only way to DROP a dependent USING is to drop the register that it depends on – register 5 in this case. Establishing addressability to `GAMESTA2` looks like this,

```
         DROP    R5
         USING   GAME,R5
         USING   STATS,GAMESTA2
```

This seems like an unsatisfactory solution. The most obvious drawback is that we can't access the statistics of both teams at the same time. An unlabeled dependent USING is fine for a single use of a DSECT when you need to define a structure within a structure, but there is a better solution when we want to use a DSECT multiple times within a larger structure.

Labeled Dependent USING Statements

By using labeled dependent USINGs, we can use a DSECT multiple times within a larger structure and minimize register usage at the same time. Here is the final solution for getting addressability to all the fields. First we establish addressability for the `GAME DSECT`,

```
         USING GAME,R5
         LA    R5,RECIN
```

Then we write labeled dependent USINGs for each instance of the `STATS` substructure,

```
G1       USING STATS,GAMESTA1
G2       USING STATS,GAMESTA2
```

Once these are defined, we can assess subfields by using the labels as qualifiers,

```
G1.HITS, G1.RUNS, G1.ERRORS, G2.HITS,G2.RUNS,G2.ERRORS
```

It's easy to end the domains of labeled dependent USINGs – simply DROP the labels like this,

```
         DROP  G1
         DROP  G2
```