Q-Type Address Constants,
Dummy External Symbols,
and
Pseudo-Registers
Q-Type Address Constants, Dummy External Symbols, and Pseudo-Registers

A common problem in the use of higher-level programming languages is how to describe the organization and use of virtual areas of memory. For example, in Fortran the compiler must be able to produce code to reference COMMON areas. The Fortran programmer writes a series of declarations which describe the positions of all variables in the virtual area (COMMON block) relative to one another; it is his responsibility to ensure that no conflicts arise due to different descriptions of the same areas in other routines using the same COMMON block. The Linkage Editor then completes the processing of such COMMON blocks by collecting together data about the lengths of each, and allocating space in the generated load module for them. Three important features of this process are: (1) the locations of variables to be stored in each COMMON area are fixed relative to each other at compile time; (2) the locations of the COMMON areas in the load module are fixed relative to one another at link-edit time; and (3) space for the areas is allocated by the Linkage Editor.

Slightly more flexibility is allowed in the Assembler Language through the DSECT statement. Data following that statement describes the layout of a virtual area of memory in which the relative locations of data in the area are again bound to one another at "compiler" time. In this case, however, the binding of the virtual areas described by DSECT statements relative to one another is at the discretion of the programmer, since he may allocate space for such areas at assembly time, at link-edit time (through use of COM, CSECT, and similar facilities), or even at execution time (by dynamically acquiring core space).

In PL/I, the availability of CONTROLLED variables or based structures means that the compiler must generate code for referencing areas of memory that are allocated at execution time; the declarations of such variables in the source program is equivalent to the DSECT facility of Assembler Language.

Up to this point, there is little problem in implementing any of these facilities, since references to data in such virtual areas may be made through a base address stored in the program at link-edit time (in the case of Fortran COMMON blocks and statically declared PL/I variables), or at execution time (in the case of dynamically allocated variables). The problem of generating code becomes more difficult when it is possible to compile segments of a program separately, since information about the virtual areas used by the routines must be preserved for combination at link-edit time. Again, this is a fairly standard procedure, as the wide acceptance of Fortran COMMON attests.

A more difficult situation arises when the code must be re-entrant. This means that all variables must be dynamically allocated, so that only relative bindings of variables into various programmer-defined virtual areas may be made during the translation phase. To take an example from PL/I, the pointers to data areas associated with blocks and procedures cannot be part of the load module if the code is to be re-entrant. The Linkage Editor must therefore provide a means of binding the virtual areas relative to one another, so at execution time the program may allocate instances of the data areas for each of the times its execution is initiated.
An example may help to illustrate the problem: suppose a program is made up of three subprograms, each of which refers to two of three virtual data areas. Routine P1 refers to A and B, P2 refers to A and C, and P3 refers to B and C. Since the actual data areas whose organization is described by A, B, and C must be allocated at execution time, it would be simplest to have the Linkage Editor produce a virtual "super-area", in which all similarly named data areas are collected and combined, and then compute offsets in the super-area for each of the component parts. The compiler must have made some provision for the Linkage Editor to store these offsets back into the program. In our example, this means that a reference to a variable AVAR in data area A requires an address made up of three components:

1) a "base" address for the super-area containing A,B, and C, and for which space was allocated at execution time;

2) the offset in this large region of the origin of the data area A, as computed by the Linkage Editor;

3) the displacement from the origin of data area A of the variable AVAR, as assigned by the compiler or language translator in which AVAR was declared.

Figure 1 below illustrates this process, in reverse order.
Because the program is to be re-entrant, the "base" address for the complete data area must be carried in a register at all times.

This simple scheme is close to the one used on System/360 by PL/I and other processors such as the SPASM Fast Assembler (ref. 1) to handle memory-reference and core allocation problems. However, the addressing structure of System/360 makes it rather difficult for the Linkage Editor to modify the addresses of instructions referring to variables in each data area with the appropriate offset of the data area within the "super-area". Thus, a somewhat different approach is used, which is more in harmony with the base-displacement addressing scheme.

It would be natural in a multi-register machine to allocate registers to point to the various data areas, and have the run-time space supervisor place pointers to each of the virtual data areas into the proper registers; then a reference to AVAR would be made with the compiler-assigned displacement, the link-editor assigned register, and the space-supervisor assigned base address. It is clear, however, that (1) we would rapidly run out of registers, since any number of virtual data areas are at least conceptually available to the programmer; (2) register allocation for calculations in the compiled code would be extremely difficult, since the compiler could not know how many registers might have to be occupied by base addresses required by other modules which may be link-edited with the one currently being compiled; and (3) some means would have to be provided for identifying the register components of those instructions which need to be allocated at link-edit time, and distinguishing them from references which do not need such adjustment, such as branch addresses and references to constants.

Instead of allocating registers and base addresses directly, a simple alternative is available: at the cost of an extra memory reference for each separate data area, we can simulate the availability of these many base registers by keeping a list of "pseudo-registers" in memory. Thus, the run-time space-allocation routine needs to allocate space for two things: first, the usual space for the virtual "super-area", and second, space for the base addresses of each of the component data areas in that super-area. Then, instead of having to avoid compiling code that might disturb the contents of many real hardware registers, the compiler need only reserve a single run-time register which will be the sacred pointer to the list of pseudo-registers. As the name implies, these pseudo-registers are usually treated as read-only areas of memory by compiled code. (They can be changed by the space manager in cases where the data areas they point to are de-allocated to conserve core space, and subsequently re-allocated.)

Unfortunately, this elegant scheme is one of the best-hidden and ill-documented features of OS/360. The terminology used in the Assembler Language manual (ref. 2) is entirely different, and gives no hint of the possible uses to which this technique may be applied. We will therefore give some examples of the use of pseudo-registers through the Assembler Language, trying to clarify what is happening at each stage.
The basic mechanism in the Assembler Language is the Q-type Address Constant, in conjunction with the DSECT or DXD declarations. For example:

<table>
<thead>
<tr>
<th>AA</th>
<th>DSECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARA1</td>
<td>DS</td>
</tr>
<tr>
<td>VARA2</td>
<td>DS</td>
</tr>
<tr>
<td>VARA3</td>
<td>DS</td>
</tr>
</tbody>
</table>

might appear in an assembly-language version of a program that intends to refer to a virtual data area named AA. The displacements of the variables VARA1, VARA2, and VARA3 are calculated relative to one another by the Assembler. To make the name AA appear as an externally-identifiable virtual data area (that is, identifiable to the Linkage Editor), the name AA is used in a Q-type address constant, e.g.

AA OFFSET DC Q(AA)

The Assembler will then identify AA as an External Dummy (XD) symbol, which has its own External Symbol Dictionary (ESD) type; additional information will be provided in that dictionary which describes the length and alignment of the virtual area described under the name AA. This latter information is necessary for the linkage editor to correctly compute and assign offsets in the super-area.

In cases where the programmer does not wish to describe the item through the use of a DSECT, another mechanism if provided. This case might arise if the datum to be described is simply an address of a data area, as would occur for the pseudo-registers of PL/I; we want merely to be able to get the pointer to the desired data area without having to say any more about the area itself. The statement used for this is the Define External Dummy statement, of which some examples follow.

<table>
<thead>
<tr>
<th>AAPTR</th>
<th>DXD</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLWDS</td>
<td>DXD</td>
<td>4D</td>
</tr>
</tbody>
</table>

The operand field of a DXD statement is the same as a single operand of a DS statement; as mentioned above, it provides the name to be identified as an External Dummy symbol, and the length and alignment associated with it.

Now that we have indicated how the Assembler generates Dummy External symbols, we will examine how they are processed by the Linkage Editor, and then come back to illustrate how they are used in Assembler-Language programs to achieve the desired results.

The linkage Editor collects from each module in its input streams all symbols identified as (and now the name has been changed from External Dummy symbols) Pseudo-Registers (PR). There is a single entry in the PR table for each symbol, containing three additional items of information besides the name: the length of the area to be allocated for the symbol, the alignment required for the area, and the offset that will (later) be assigned to that symbol. As the symbols are entered in the table, any duplicate entries for which a name already exists in the table are given the greater length and more restrictive alignment of the old and new entries; this guarantees that adequate space will be allotted, and on proper boundaries, for each of the areas desired.
When the end of the input is reached, the Linkage Editor makes a pass over the table of XD/PR symbols, and assigns sequential offsets to each item so that the length and alignment restrictions are satisfied. These offsets are then the values of the symbols, and may be used to refer to the super-area described by the collection of XD/PR symbols.

To make use of the data provided by the Linkage Editor in this table, the Assembler Language programmer uses the Q-type address constant. For example, in the statement named "AAOFFSET" above, the Linkage Editor would place the value of the offset of the virtual area AA into the fullword address constant AAOFFSET. This could then be used by the programmer as he desires.

To return to the earlier example of three programs which refer to three different data areas, suppose we are working in routine PI which refers to variables in data areas A and B, and that we must add a variable VARB from B to the variable VARA in A. Then we could write code like that given in Figure 2.

```
A DSECT  } describe virtual data area A
    ___
VARA DS    F
    ___
B DSECT  } describe virtual data area B
    ___
VARB DS    F
    ___
PI CSECT  
    ___
    start of programs; assume that register 12 contains the address of the super-area containing all data
    ___
L 2,AAOFFSET get offset of data area A
AR 2,12     add base of super-area
USING A,2   now have true base of A
L 3,BOFFSET same for B
AR 3,12
USING B,3
L 0,AVAR
A 0,BVAR
ST 0,AVAR
DROP 2,3   
             or whatever processing is desired
AAOFFSET DC Q(A) offset of A, provided by Linkage Editor
BBOFFSET DC Q(B) same for B
```

Figure 2.
While this example allows for the most general use of External Dummy symbols, the process of constructing base addresses by adding the offset to the base address may require time and space that can be saved in certain cases. In SPASM and PL/I, a slightly different method is used: rather than calculate the addresses of the data areas as and each time they are needed, the space allocator calculates the address of the data area and stores it in the list of "pseudo-registers". The programmer then refers not to the data areas by the use of Dummy External symbols, but to their addresses: the offsets in the pseudo-register "vector" (or FRV, as it is called in PL/I) now are those appropriate to single addresses rather than entire data areas. Because the offsets are smaller, they can be used in a clever method to reduce the number of additional instructions to one, namely that needed to load the desired address.

Suppose that the data areas described by DSECTs A and B are as before, but now we may assume that the External Dummy symbols ADDRA and ADDRB will have values which are the offsets of the desired addresses in the pseudo-register vector. Then we could recode the program of Figure 2 as shown in Figure 3.

It can be seen that this technique requires two independent things. First, the instruction which references the data from the super-area, or FRV, must be of RX type, so that the digit which specifies the sacred register pointing to the FRV may be placed in the index position of the instruction (or otherwise, an appropriate LA instruction should precede the non-RX instruction); and second, the offsets generated for the symbols ADDRA and ADDRB must be less than 4096 or else there will be spillover into the base-digit field of the instruction. (The programmer can of course provide for this by having an address in register 1 which is fixed at a constant value of 4096.) Because the technique chosen here will generally specify small entries in the super-area, this problem rarely arises.

```
   81   53   56   66   66   76
   A     A     R12 is sacred register

   ADDRA   DXD  A  }  must appear before use in Q-constants
   ADDRB   DXD  A

   L   2,0(12,0)    R12 in index digit
   ORG  *-2            back up 2 bytes
   DC   QL2(ADDRA)  put offset into displacement
   USING A,2
   L   3,0(12,0)    same for pointer to B
   ORG  *-2
   DC   QL2(ADDDB)
   USING B,3
   L   0,VARA
   A   0,VARF
   ST   0,VARA
   DRJMP 2,3
```

Figure 3.
As noted in the commentary on the DXD's in Figure 3, the DSECT or DXD statements which define Dummy External symbols must appear BEFORE they are used in the Assembler-Language program. There is no good reason for this; it is simply an implementation restriction.

The careful reader may have one nagging question left at this point (and no more, it is hoped): how do you find out how much space to allocate for the super-area described by the collected Dummy External symbols? Because there is no way for the programmer to know which of the items in the Linkage Editor's PR list was last, there is no direct means of using a Dummy External symbol to determine how long the area was. (This is because the Linkage Editor uses no sorting or aligning algorithm in calculating offsets for Dummy External symbols.) The desired piece of data is provided by the CXD instruction, which reserves a fullword for the cumulative length of the super-area. The Linkage Editor is informed by another ESD entry (PR type 2) of the location that requests the total length, and fills it after all the offsets have been computed. On the link-edit map, this is indicated by the line "Location xxxxxxx requests cumulative pseudo-register length".

There are still areas where the facilities provided by Dummy External symbols could be expanded or improved. For example, space could be saved in the super-area (PRV) by having the Linkage Editor sort the table entries by length and alignment, so that there would be a minimum of gaps in the allocated space. Secondly, because the facility provided by the use of Dummy External symbols is so powerful, it seems a pity that there aren't classes of dummy symbols, so that one could describe many, rather than one, virtual dummy sections. This would allow programmers to make greater use of the symbolic capabilities of the Assembler and Linkage Editor, and thereby write more general and widely applicable programs.

References
