Published: 5/5/67

Identification

Appendix to Section BN.9.01.

James F. Gimpel

Purpose

To provide supplementary and/or formal information on the following topics

1. directly-addressable data
2. packed data
3. arrays and multipliers

Directly-Addressable Data

Definition: A directly-addressable data item is one whose location with respect to the first physical location of its outermost containing aggregate is invariant with respect to any adjustable array bound or string length.

Definition: An adjustable element (scalar, structure, or array) is one whose declaration contains an adjustable array bound or an adjustable string length.

Definition: Any element (scalar, structure or array) not contained within an array is said to have known beginning if the element is on level 1 (i.e., is the outermost structure) or if the element is on level i and

(a) the containing structure on level i-1 has known beginning, and

(b) all preceding elements on level i are not adjustable.

Remark: A data item is directly addressable if and only if

(a) it has known beginning, or

(b) it is a member of a nonadjustable array, having known beginning, or

(c) it is a member of an adjustable array having known beginning whose only adjustable value is the first upper bound.

The Packing Algorithm

A string or an array of strings is packed or unpacked according to the following algorithm. Consider the declaration of the outermost aggregate A containing the string. Remove all dimension attributes. What remains is either a structure or a scalar; call it A'. Within A', if every scalar within any structure B' is a nonvarying bit string or if every scalar within B' is a nonvarying character string, then B' is packed and is not packed otherwise. A string is packed or unpacked according to whether its immediately encompassing structure is packed or unpacked. A string not contained in a structure is not packed.

An array of strings or an array of structures or a string or a structure in A is packed according to whether the corresponding string or structure is packed in A'.

Example: Declare alpha (10) bit (5); is not packed because a string not contained in a packed structure is not packed.

Example: declare 1 alpha (10), 2 beta bit (5);

is packed because we now have a structure.

Example: declare 1 alpha, 2 beta (10) bit (5);

is also packed for the same reason.

```
Example: declare 1 alpha,
                   2 beta  char (1),
                   2 gamma bit (5),
                   2 delta bit (12);
```

is not packed.

```
Example: declare  1 alpha (0:5),
                   2 beta (0:100),
                        3 gamma bit (10),
                        3 delta bit (1),
                   2 epsilon (0:49) char (1);
```

Alpha and epsilon are not packed but beta is. If epsilon were a bit string, alpha and epsilon would be packed.

Arrays and Multipliers

Consider the array: dcl alpha (-2:0, 3:4) fixed;

The elements of alpha are allocated into contiguous core memory as

b	alpha (-2,3)
b+1	alpha (-2,4)
b+2	alpha (-1,3)
b+3	alpha (-1,4)
b+4	alpha (0,3)
b+5	alpha (0,4)

where b is the base of the aggregate alpha (i.e., the first physical location). When a member of alpha is addressed at runtime, say alpha (i_1,i_2) the computation of the address could proceed as

$$b + (i_2-3) + (i_1+2) \times 2.$$

For example, the address of alpha (-1,3) is

$$b + (3-3) + (-1+2) \times 2 = b+2$$

which checks.

More generally, let alpha be declared as

declare alpha $(\ell_1:u_1, \ell_2:u_2,\ldots,\ell_n:u_n)$
attribute-list;

The attribute list will govern the size, s, of an element of alpha. The size is measured in bits if alpha is packed and in words if alpha is unpacked. Alpha may be a structure in which case s will measure the size of the structure. The multipliers for the array are defined as:

$$m_n = s$$
$$m_{n-1} = (u_n-\ell_n+1)s$$
$$m_{n-2} = (u_{n-1}-\ell_{n-1}+1)m_{n-1}$$
$$\vdots$$
$$m_1 = (u_2-\ell_2+1)m_2$$

The address of alpha $(i_1,i_2,\ldots,i_n)$ is computed as

$$b + \sum_{j=1}^{n} (i_j-\ell_j)m_j$$

which becomes

$$b + \sum_{j=1}^{n} i_j m_j - \sum_{j=1}^{n} \ell_j m_j.$$

The virtual origin v is defined as

$$v = b - \sum_{j=1}^{n} \ell_j m_j.$$

It is the address we would obtain by setting the subscripts equal to 0. It serves as a convenient base from which to compute any address; that is the address of any array element becomes

$$v + \sum_{j=1}^{n} i_j m_j.$$

Normally, the multipliers and the offset to the virtual origin from b are kept in the array dope. For directly addressable data items these quantities are known at compile time and code can be generated which ignores the dope.