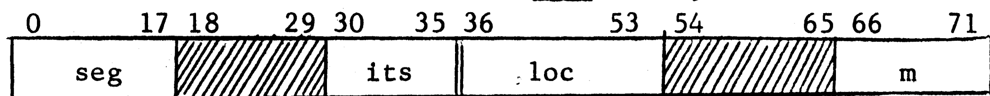
TO: MSPM Distribution
FROM: R. M. Graham
SUBJ: Pointer Data
DATE: 02/09/68

The attached section (BB.2.02) for the most part gathers together the definitions of the Multics standard data types and their identification codes which appear in sections BB.2, BD.1.00, BP.2.01, and BP.2.02. The format for short varying strings, soon to be implemented in EPL, is defined here. It also expands the definition of pointer data. In addition, both argument list elements and specifier elements are defined to be pointer data, thus expanding their format. BB.2.02 supersedes those portions of BB.2 which deal with data type representation. The remainder of BB.2 will someday be superseded by BB.2.01, User Interfaces with the Multics System, and BB.2.03, Intra-System Module Interfaces.

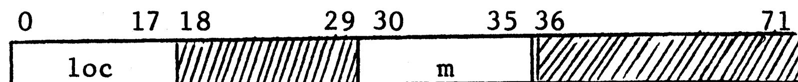
There are now three subclasses of pointer data:

i) External: This is the old its form,



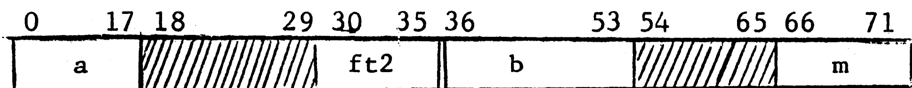
where seg is the segment number and loc is the location within seg.

ii) Internal: This is the new form,



where the segment number, seg, is understood to be the segment in which this internal pointer resides and loc is the location within seg.

iii) Link: This is the form used in the linkage section,



where both the segment number, seg, and the location within it, loc, are defined by auxiliary information pointed to by a and b. This is usually accomplished by invoking the linker (either via the ft2 fault or by direct call: see BD.7.04).

It should be noted that the only way to be certain of accessing correctly a pointer datum, without programmed examination of the datum, is to use the instruction,

```
eapbp       datum,*
```

Use of the instructions (suggested in BD.7.02)

```
ldaq        datum

staq        add
```

to move a pointer datum will no longer work correctly in all cases. Since link type pointer data contains a self relative quantity (<u>a</u>) it may not be moved at all. Internal type pointer data may not be moved out of its segment. Use of the new sequence,

```
eapbp       datum,*

stpbp       add
```

will of course force any link involved. To allow movements without forcing links see the further comments in BB.2.02. All three pointer types may also contain a modifier, <u>m</u>, indicating further indirection: see BB.2.02 for further details.

Argument lists for a standard call and specifiers are both composed of pointer data. Hence, the above comments also apply to argument lists and specifiers. Section BD.7.02 and other relevant MSPM sections will be revised in the near future to reflect this enlargement of pointer data.

Published: 02/09/68

Identification

Multics Standard Data Types
R. M. Graham

Purpose

This section specifies the Multics standard data types and defines their representation in the GE 645. All arguments for normal user interfaces with the system must be limited to data types from this set. Thus a language translator producing programs to execute in Multics need know how to handle only a small number of rather simple data types.

Introduction

The following topics will be discussed in the order in which they are listed:

1. A listing of Multics standard data types and their identification codes.

2. Elements of an argument list.

3. Representation and detailed discussion of each non-string scalar.

4. The representation of strings, general comments about free storage (for varying strings), and substitution rules for strings.

5. Specifiers

6. Dope

7. Accessing of array elements.

8. Representation and detailed discussion of varying and non-varying string scalars, arrays of non-string scalars, and arrays of varying and non-varying strings.

In the following discussion a "word" is defined to be a 36-bit, GE 645 machine word. A "word-pair" is defined to be a pair of contiguous 36-bit, GE 645 machine words, the first of which is located at an even memory address.

Summary of Data Types

The Multics standard data types and their identification codes are listed below. The identification code is used in calls (see BD.7.02) and in symbol tables (see BD.1.00).

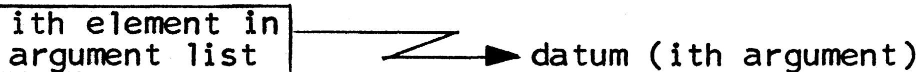
ID Code	Data Type
1	single-word integer
2	double-word integer
3	single-word floating-point
4	double-word floating-point
5	single-word integer complex
6	double-word integer complex
7	single-word floating-point complex
8	double-word floating-point complex
9	non-varying bit-string
10	long varying bit-string
11	non-varying character-string
12	long varying character-string
13	pointer data (external, internal, and link)
14	offset data (offset from a pointer datum)
15	label data
16	entry data
17	1-dimensional array of type 1
18	1-dimensional array of type 2
19	1-dimensional array of type 3
20	1-dimensional array of type 4
21	1-dimensional array of type 5

ID Code	Data Type
22	1-dimensional array of type 6
23	1-dimensional array of type 7
24	1-dimensional array of type 8
25	1-dimensional array of type 9
26	1-dimensional array of type 10
27	1-dimensional array of type 11
28	1-dimensional array of type 12
29	1-dimensional array of type 13
30	1-dimensional array of type 14
31	1-dimensional array of type 15
32	1-dimensional array of type 16
39	short varying bit string
40	short varying character string
41	1-dimensional array of type 39
42	1-dimensional array of type 40

Argument List Elements

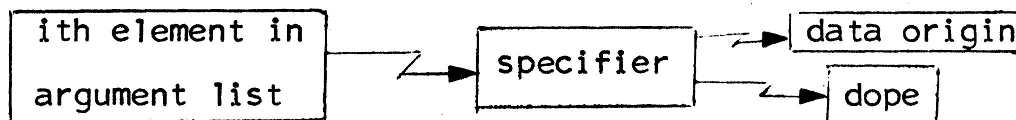
The Multics standard call is defined in BD.7.02. The argument list described in that section consists of a list of pointer data. What each argument list element is actually pointing to depends upon the data type of the corresponding argument. There are two cases:

i) Datum is a non-string scalar (types 1-8 and 13-16 above): In this case the argument list element points directly to the datum. If the datum occupies more than one word the element points to the first word of the datum.



ii) Datum is a string scalar (types 9-12, 39-40) or any 1-dimensional array (types 17-32, 41-42): In this case the argument list element points to a _specifier_ which in turn points to the datum (or to an area in which the datum is found) and to _dope_ for that datum.

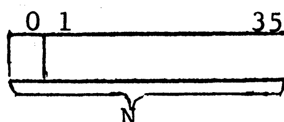
A specifier contains two or more its pairs, hence, it is process dependent. Dope contains information describing the datum and is process independent. The datum is located by using both the specifier and the dope.



Non-String Scalars

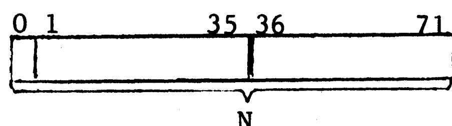
The following is a description of the representation of each of the non-string scalar data types. The identity of each type preceeds its description.

1) Single-word integer: Stored in a single word in the GE 645 hardware format for a single-word fixed-point number,



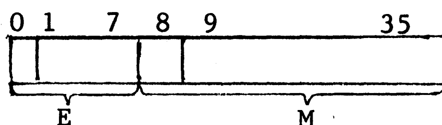
where N is a signed integer in 2's complement form.

2) Double-word integer: Stored in a word-pair in the GE 645 hardware format for a double-word fixed-point number,



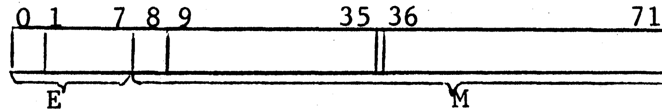
where N is a signed integer in 2's complement form.

3) Single-word floating-point: Stored in a single word in the GE 645 hardware format for a single-word floating-point number,



where both E and M are signed integers in 2's complement form. E is the exponent (base 2) and M is the mantissa.

4) Double-word floating-point: Stored in a word-pair in the GE 645 hardware format for a double-word floating-point number,

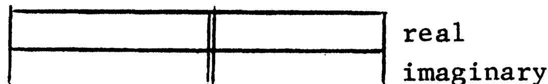


where E and M are signed integers in 2's complement form. E is the exponent (base 2) and M is the mantissa.

5) Single-word integer complex: Stored in a word-pair,

where each word is a single-word integer, the first word is the real part, and the second word is the imaginary part.

6) Double-word integer complex: Stored in two consecutive word-pairs,

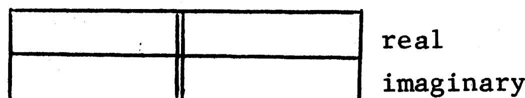


where each word-pair is a double-word integer, the first pair is the real part, and the second pair is the imaginary part.

7) Single-word floating-point complex: Stored in a word-pair,

where each word is a single-word floating-point number.

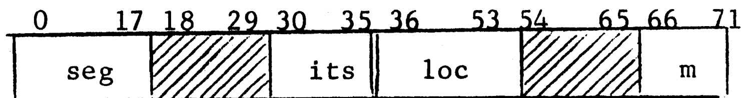
8) Double-word floating-point complex: Stored in two consecutive word-pairs,



where each word-pair is a double-word floating-point number.

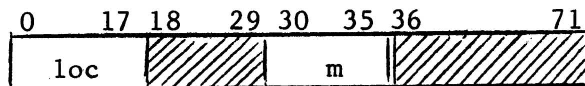
13) Pointer: Stored in a word-pair. A pointer datum defines, directly or indirectly, a GE 645 machine address. The indirection may be more than one level, i.e., the pointer datum may be an arbitrarily long indirection chain. Let seg be a segment number and loc be a location within that segment, where both seg and loc are unsigned 18-bit integers. In addition, let m be an indirect modifier. Then a pointer datum consists of a chain which is a mixture of the following three types of elements all of which contain an indirect modifier, except for the last one. Stated another way, a pointer datum is such that the machine address which it defines may be obtained by executing a single effective address type instruction.

i) External:



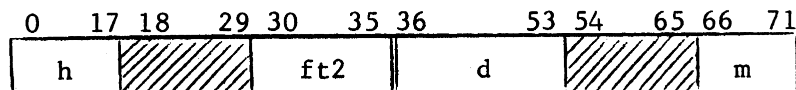
where both seg and loc are explicitly stated and m is the indirect modifier (if any). The shaded part of the word-pair is ignored, but should, in general, be zero.

ii) Internal:



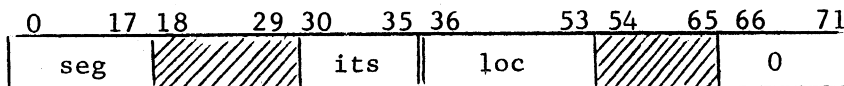
where loc is explicitly stated and seg is the number of the segment in which this datum resides.

iii) Link:



where h and d lead to descriptive information which defines both seg and loc. This definition is usually accomplished by the linker, either because of an ft2 fault or by a direct call to the linker: see BD.7.04.

As an aid to understanding pointer data and its manipulation we will call,



the _normalized_ form of a pointer datum. This form, since the modifier is zero, directly defines the machine address (_seg_, _loc_). The most important characteristic of all three types of pointer data is the following: If _datum_ is the location of a pointer datum, then executing the instructions,

```
eapbp    datum,*

stpbp    temp
```

will place in the word-pair at _temp_ the normalized form of the original pointer datum. Use of the pair of instructions above is the simplest, foolproof method of moving pointer data. It has the disadvantage, however, that any links in the chain of indirect word-pairs are established. A link is established by invoking the linker. The linker replaces the link by an external type pointer (_its_). In order to obtain the segment number to put in the _its_, the segment must be located in the file system, even if it is never actually referenced.

If the premature establishment of links is considered to be too costly, one of two alternative methods for moving pointer data may be used. The first alternative is to construct an indirect external pointer to the pointer datum and move it. For example,

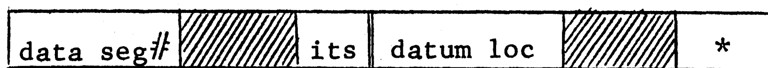
```
eapbp    datum

stpbp    temp

lda      =Ø20,dl

orsa     temp+1
```

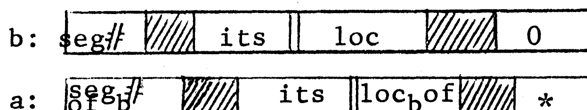
The contents of _temp_ will then be,



A disadvantage of this method is that comparison of pointer values will give unexpected results. For example, suppose that a and b are both pointer type variables, then after the substitution,

a = b

the contents of _a_ are not equal to the contents of _b_! Rather, _a_ points to whatever _b_ points to by means of pointing to _b_ with an indirection modifier:



Another disadvantage of this alternative is that the indirect chain increases in length each time the pointer datum is moved.

The second alternative is to test for the type of pointer and only if it is not external build the indirect external pointer. The following code sequence will test to see if datum contains an external pointer,

```
lda     datum
ana     =Ø77,dl
cmpa    =Ø43,dl
tze     extptr          transfer if external pointer
```

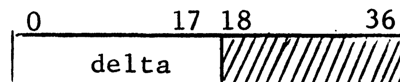
If the datum is an external pointer it is moved with the instructions,

```
ldaq    datum
staq    temp
```

Using this alternative, the beginning of the indirect chain is always an external pointer after the first move of the datum. Hence, the chain will not grow in length on successive moves.

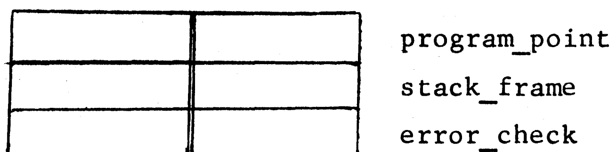
External pointer data is process dependent since its representation includes a segment number. On the hand it is not location or segment dependent, hence, it may be moved about freely. Internal pointer data is segment dependent, hence, it may not be moved out of its segment. Link pointer data is location dependent and may not be moved at all. However, both internal and link pointer data are process independent, which is their principal advantage since they may be generated at compile or assembly time, whereas external pointer data must be generated during execution.

14) Offset: Stored in a single word,



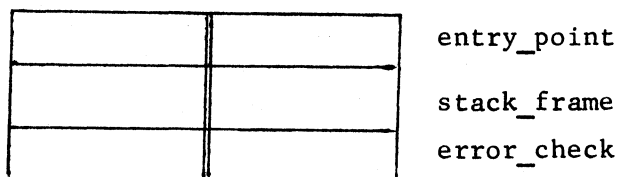
where delta is an offset relative to some pointer. An offset is process independent. The connection between an offset datum and the pointer for which it is an offset is a function of the procedure using the datum.

15) Label: Stored in three consecutive word-pairs,



where the first two word-pairs are both pointer data (type #13) and the last word-pair has not yet been defined. Label data is usually used to communicate an abnormal return point to some other procedure: see BD.9.05. Program_point is the address (recall that pointer data defines a machine address), <u>in the linkage section of the procedure in which the label was defined</u>, of instructions which will re-establish the correct value of the linkage pointer (lb←lp base pair) and transfer to the proper location within the procedure: see BD.7.01, BD.7.02, and BD.7.03. stack_frame is the machine address of the base of the stack frame in use at the time when the label datum was defined, i.e., the contents of the sb←sp pair at that time. error_check may someday contain error checking information.

16) Entry: Stored in three consecutive word-pairs,



where the first two word-pairs are both pointer data (type #13) and the last word-pair has not yet been defined. Entry data is usually used to communicate a procedure entry point to some other procedure. entry_point is the address, in the linkage section of the procedure for which this datum is an entry point, of instructions which will establish the lb←lp base pair and transfer to the proper location in the procedure: see BD.7.01, BD.7.02, and BD.7.03. stack_frame is either a null pointer if entry_point is an external procedure or the address of the base of the stack frame in use at the time when the entry datum was defined if entry_point is an internal procedure: see BD.7.02.

Representation of Strings

There are two kinds of strings: bit and character.

i) Bit Strings: An n-bit string may begin at any bit position within a word and extends into as many consecutive following words as are required to contain the string. All words, except possibly the first and last, contain 36 bits of the string. The first (leftmost) bit of the string is bit 1, while the last (rightmost) is bit n.

ii) Character Strings: Individual characters are coded in 7 bits, as specified in BC.2.01, right justified in 9-bit bytes with leading zeros. An n-character character string is represented as if it were a 9*n-bit bit string, except that it must start at bit position 0, 9, 18 or 27 within a word. The first character of the string is character 1 and the last is character n.

There are two classes of strings: varying and non-varying.

i) Non-varying strings: A non-varying string has a fixed length and remains in a fixed memory position throughout the scope of its definition. When substituting a string x, of length lx, for the contents of a non-varying string y, of length ly, the following rules apply,

   a) if lx < ly; a copy of the string x is extended (padded) on the right until its length is ly and then substituted for string y. The padding is zeros for bit strings and blanks for character strings.

   b) if lx > ly; a copy of the string x is truncated on the right so that its length is ly and then substituted for string y.

A non-varying string may be a substring of a longer string extending either to the right or left or in both directions. Hence, substitution into a non-varying string must not change any bits on either side of the string.

ii) Varying strings: A varying string has a fixed maximum length throughout the scope of its definition; however, both its length and memory position may vary during this time. The substitution rules for varying strings are slightly more complicated than those for non-varying strings. When substituting string x, of length lx, for the contents of a varying string y, of maximum length my, the following rules apply,

a) if lx > my; a copy of the string x is truncated on the right and substituted for string y so that its length is my and the length of the updated y is my.

b) if lx ≤ my; a copy of the string x is unmodified, i.e., _not_ extended in length and substituted for string y; the length of the updated y is lx.

Hence, the length of a varying string may change each time its value changes, while the length of a non-varying string never changes when its value changes. A varying string is never a substring of a longer string (in the sense of sharing memory locations). When the value of a varying string changes the memory location of the string may need to change. There are two cases:

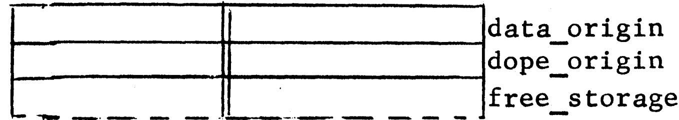
a) The new value of the string is either shorter or its increase in length is not large enough to require additional memory words for its storage. In this case the same memory locations may be reused (note however, EPL will reuse the same memory locations for short varying strings, while for other varying strings EPL will _never_ reuse the same memory locations).

b) The new value of the string is enough longer than the old value that additional memory words are required. Since, in general, words on either side of the memory location of the old value may be in use, new memory locations will have to be used for the new value. To facilitate this storage management problem all non-short varying strings must be stored in some free storage area. Four library procedures exist to carry out the bookkeeping required in the management of a free storage area. These procedures are: area_man$initial which initializes a free storage area, area_man$extend which extends a free storage area, free_man$allocate which allocates space in a free storage area, and free_man$free which returns space in a free storage area to a list of available space. These procedures and the free storage management algorithm are described in BY.16.01.

Finally, string data may be packed or unpacked (aligned). Varying string scalars and arrays of varying strings are never packed. An unpacked string scalar or element of an array of unpacked strings _always_ begins at bit position 0 within the word. Each element after the first of an array of packed non-varying strings begins at the next bit position immediately following the last bit of the preceeding element. Stated another way, unpacked strings always begin on word boundaries while packed strings need not. There are no unused bits between elements of an array of packed strings.

Specifiers

A specifier consists of two or three contiguous word-pairs,

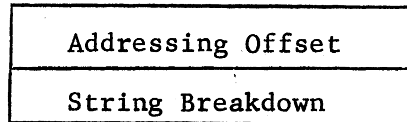


all of which are pointer data (type #13). The third pointer appears only in specifiers for non-short varying strings. Data_origin is usually the address of the first word of data (although not always: see the next paragraph on dope). dope_origin is the address of the beginning of the dope. If the datum is non-short varying string free_storage is the address of the base of a free storage area in which the datum is located. In this case data_origin is the address of further information which locates the datum in the free storage area.

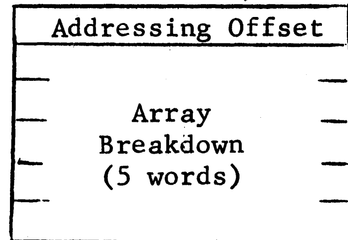
Dope

There are three major formats for dope,

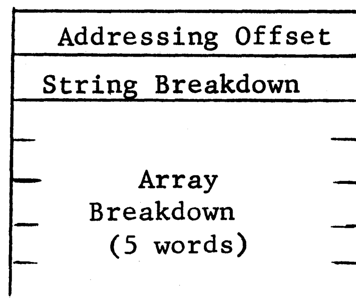
i) string scalar



ii) 1-dimensional array of non-string scalars

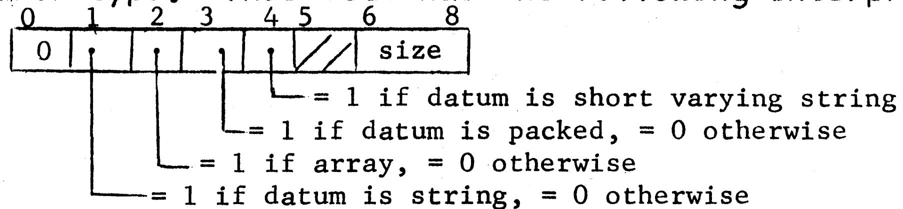


iii) 1-dimensional array of string scalars



The addressing offset is a single word and contains an offset relative to the data origin given in the specifier. Its interpretation differs depending on the data type.

The first word of any breakdown contains a code, id, in the leftmost nine bits (bits 0-8) which partially identifies the data type. This code has the following interpretation,



where size is the size in words of the elementary data item,

size = 0 for non-varying strings and short varying strings (types 9, 11, 25, 27, 39-42)

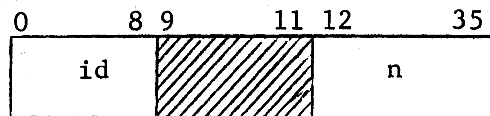
= 1 for single word non-strings (types 1, 3, 14, 17, 19, 30)

= 2 for double word non-strings and long varying strings (types 2, 4, 5, 7, 10, 12, 13, 18, 20, 21, 23, 26, 28, 29)

= 4 for four word non-strings (types 6, 8, 22, 24)

= 6 for six word non-strings (types 15, 16, 31, 32)

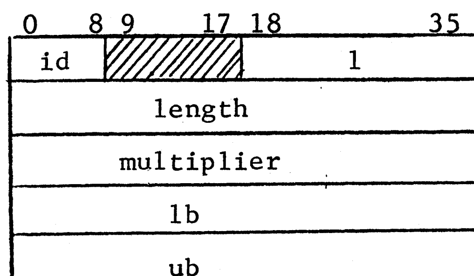
A string breakdown has the format,



where n is the length for a non-varying string or the maximum length for a varying string. The length and maximum are always expressed in bits.

Accessing of Array Elements: Array Breakdown

An array breakdown has the format,



length is the smallest number of contiguous words (bits if the array is packed) which will contain the array. multiplier is the separation, in words (in bits if the array is packed), between two elements whose subscripts differ by exactly one. It must be at least as large as the length of the elementary data item (size from id), however, it may be larger, i.e., data elements need not be contiguous in memory, but they must be evenly spaced. lb is the lower subscript bound and ub is the upper subscript bound, i.e., the elements of the array, A, are: A(lb), A(lb+1), ..., A(ub).

The following formulae are used for computing the machine address of the first word of an element, A(i), of an unpacked array of scalars,

address of A(0) = [data_origin + addressing_offset] mod 2**18

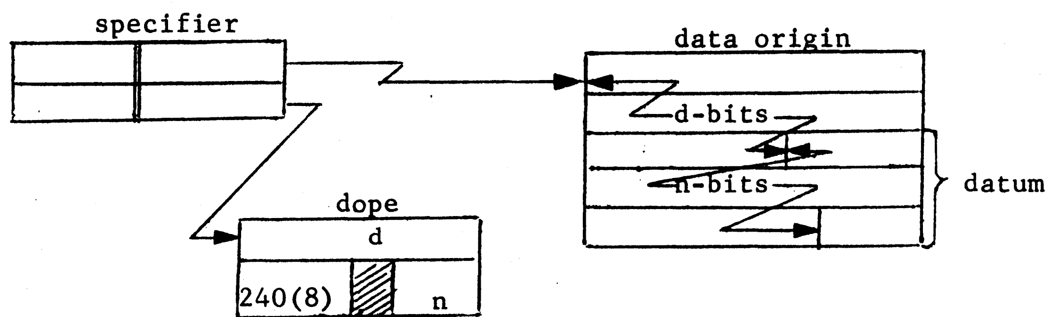
address of A(i) = [address of A(0) + delta] mod 2**18

where, delta = [i * multiplier] mod 2**18

and data_origin comes from the specifier and addressing_offset and multiplier from the dope. For all packed arrays of scalars, the formulae for computing the address of A(i) are the same as above except that delta = integer part of ([i * multiplier] mod 36*2**18).

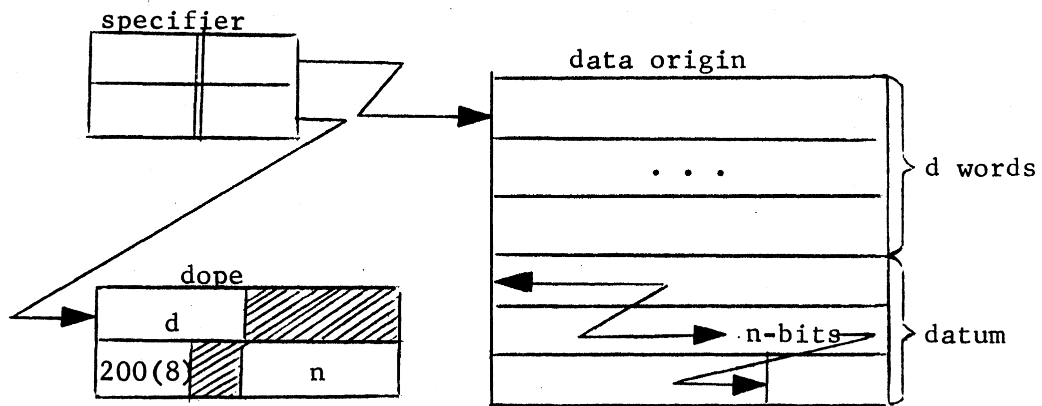
Non-Varying String Scalars (Types 9 and 11)

The layout of specifier, dope, and datum for a packed string is,



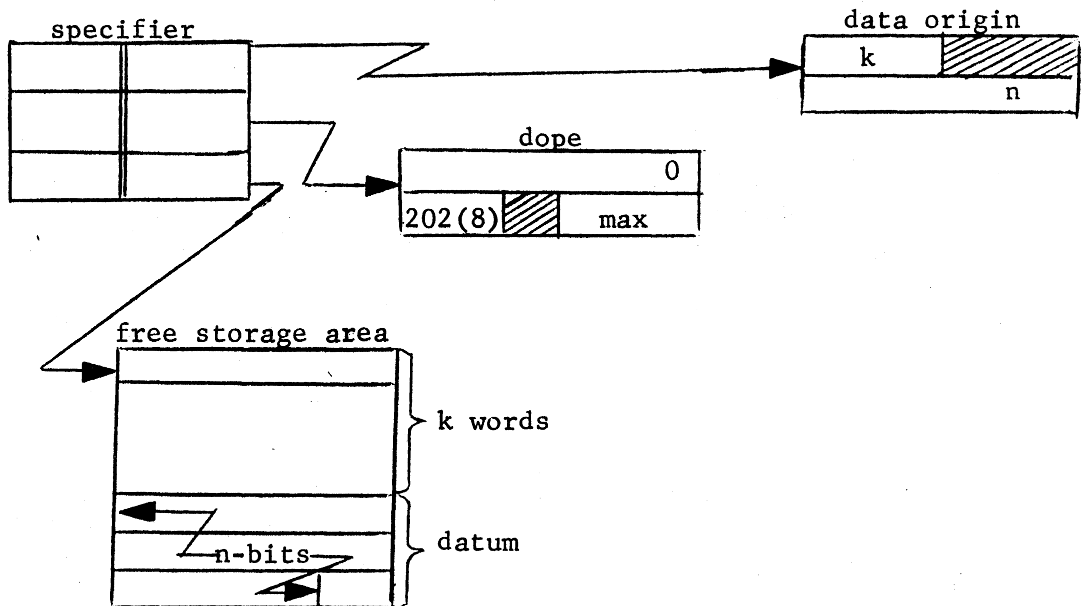
where d is the offset, 240(8) is the id code, and n is the length of the string.

Both n and d are expressed in bits. If the string is unpacked n is still expressed in bits but d is measured in words and has the form,



Long Varying String Scalars (Types 10, 12)

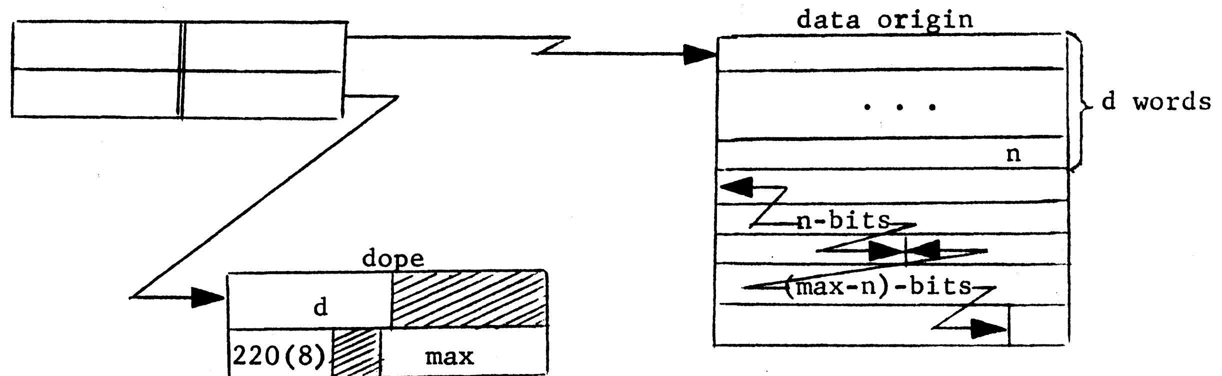
The layout of specifier, dope, and datum is,



A long varying string datum is always unpacked and is located in a free storage area, while auxiliary data, which specifies the position of the string in the free area, is found at the data origin. The addressing offset in the dope is always zero.

Short Varying String Scalars (Types 39, 40)

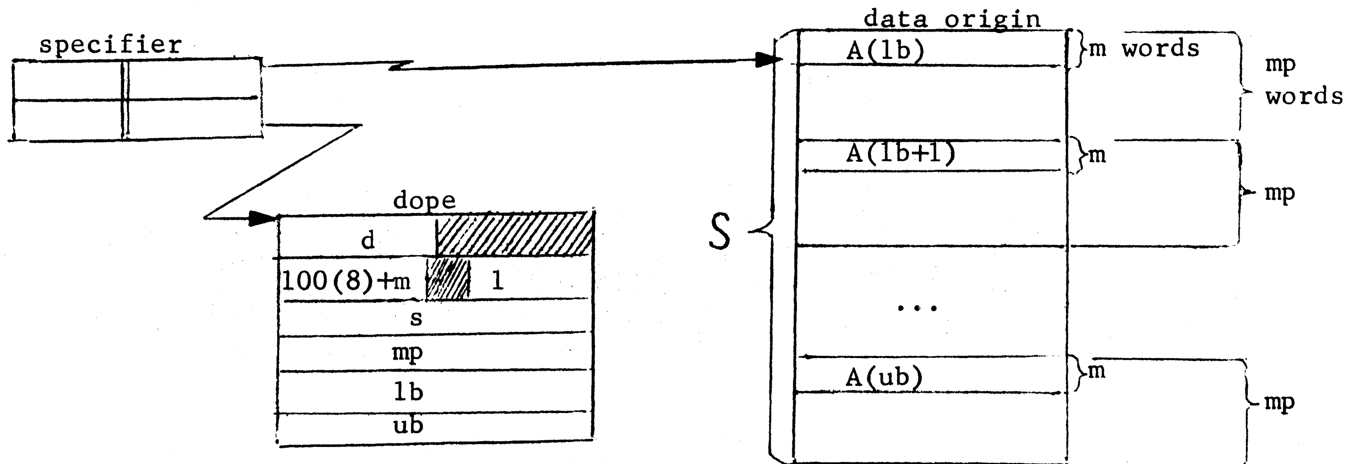
The layout of specifier, dope, and datum is,



where d is the offset (in words), 220(8) the id code, max the maximum length of the string (in bits), and n the current length (in bits). A number of memory locations sufficient to contain a string of maximum length is initially allocated d words from the data origin. The string always begins at bit 0 in the word at data_origin+d. The word immediately preceeding this (data_origin+d-1) contains the current length of the string.

1-Dimensional Arrays of Non-String Scalars (Types 17-24, 29-32)

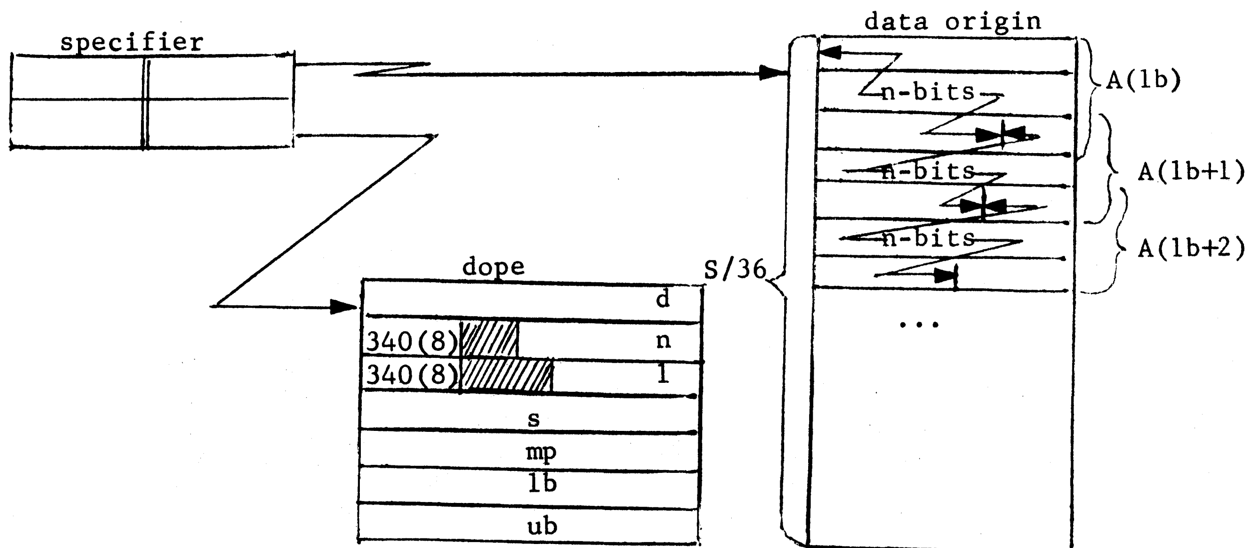
The layout of specifier, dope, and datum is (supposing the name of the array is A),



where d is the addressing offset, m (i.e., the low order octal digit of the ID code) the size of the elementary data item, s the length of the array in words, mp the multiplier, lb the lower subscript bound, and ub the upper subscript bound. The address of A(0) = data origin + d. If lb > 0, A(0) does not actually exist and d is negative (in 2's complement form). If lb < 0 then d >0. In fact, if ub < 0, again A(0) does not actually exist. Of course if lb = 0, d = 0. Even though A(0) may not exist, the address of A(0) is used as a base for computing the subscripts of the elements of A. The address of A(i) = address of A(0) + i*mp.

1-Dimensional Arrays of Non-Varying Strings (Types 25, 27)

If A is a packed array, the layout of the specifier, dope, and datum is,

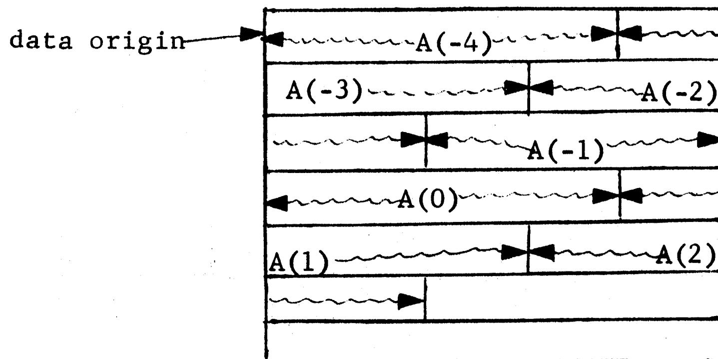


where d is the addressing offset expressed in bits, n the length in bits of each string in the array (all strings must be the same length), s is the length of the array in bits, mp the multiplier in bits, lb the lower subscript bound, and ub the upper subscript bound. The first element in the array, A(lb), always begins at bit 0 within the word at the data origin. The location of A(i) is computed by determining k = i-lb. The string A(i) then starts at bit k*n counting from bit 0 within the data origin.

As an example suppose A is a packed array of non-varying character strings, each 3 characters long, with subscript range (-4, 2). The dope for A would look like,

	108
340(8)	27
340(8)	1
	189
	27
	-4
	2

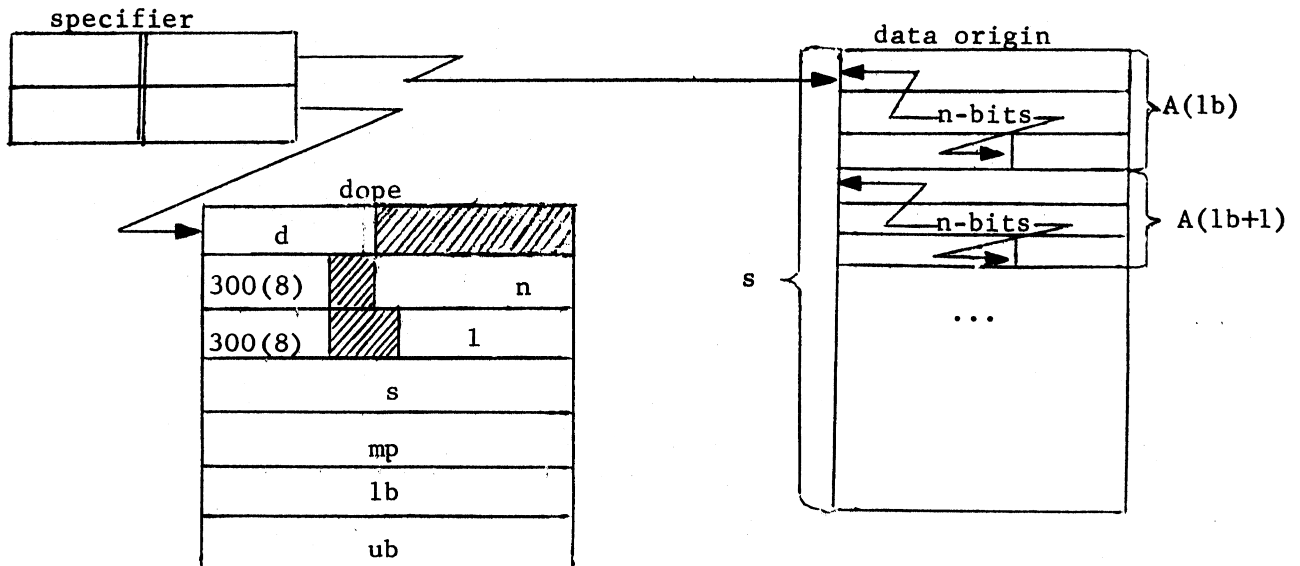
and the datum looks like,



To locate A(-2),

add A(0) = data_origin + 108
delta = (-2) * 27 = -54
add A(-2) = data_origin + 108 - 54 = data_origin + 54
which is bit 18 in first word after data_origin.

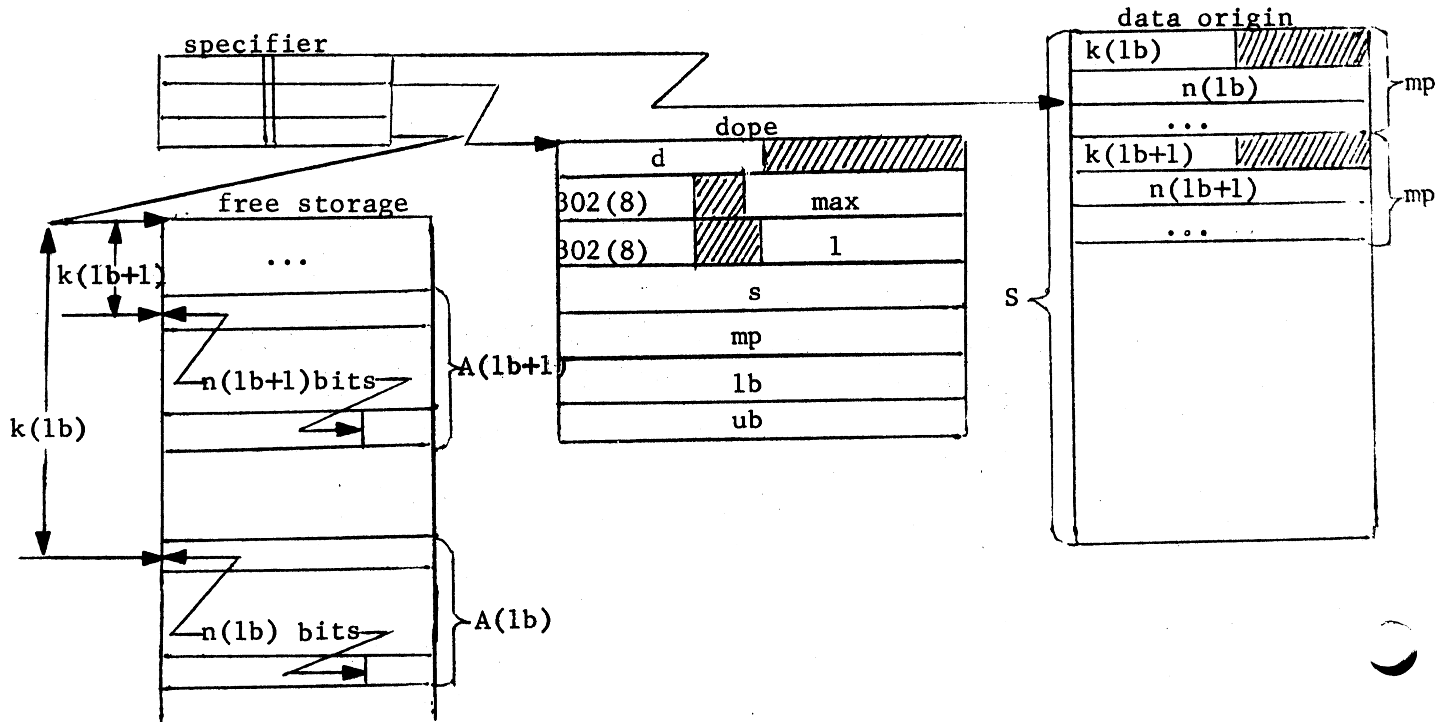
If A is an unpacked array the layout of the specifier, dope, and datum is,



where d, s, and mp are expressed in words and each string begins at bit 0 within the word.

1-Dimensional Arrays of Long Varying Strings (Types 26, 28)

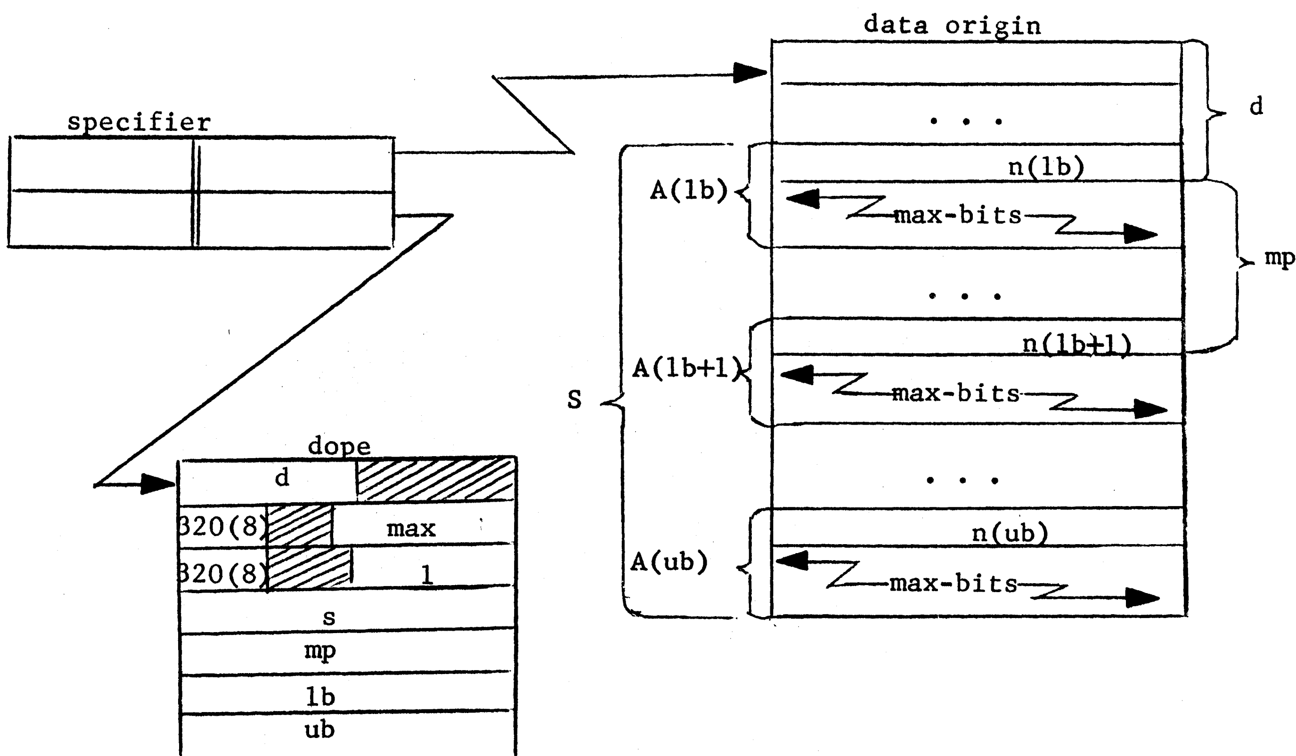
The format of specifier, dope, and datum is (supposing the name of the array is A),



where d is the addressing offset and max the maximum length of strings in the array A. The array breakdown applies to the array of auxiliary information beginning at the data origin and s is its length, mp the multiplier, lb the lower subscript bound, and ub the upper subscript bound. The auxiliary array is composed of two word elements, one element for each varying string in the array A. The ith auxiliary element specifies the current length, n(i), and location relative to the base of free storage, k(i), of the ith varying string A(i), in the array A. The strings in free storage need not be in any predictable order and may be located anywhere in the free storage area.

1-Dimensional Arrays of Short Varying Strings

The format of specifier, dope, and datum is (supposing the name of the array is A),



The dope specifies a 1-dimensional array which looks (except for the id code) like an array of unpacked, non-varying strings of length max. The current length of the string A(i) is stored just preceeding the string, n(i) in the diagram. Each length may be different, however, each must be ≤ max.