

The Segment Symbol Table design specified by MSPM Section BD.1 has been scrapped; the only remnant is the symbol header which remains as described in MSPM Section B.D.1.00A. The new design uses a fixed format rather than the variable format allowed by the old design. The new form of the Segment Symbol Table is more general, however, in that it allows for the encoding of information that is known only during the execution of the program corresponding to the Segment Symbol Table (for example, the size of an adjustable character string.)

PURPOSE

The PL/1 compiler produces an Segment Symbol Table in order to support source language debugging and data directed input/output. An Segment Symbol Table will be generated whenever:

- 1) the "table" option is used at the time the program is compiled. This causes a "full" Segment Symbol Table to be generated. All the variables, labels, and entries referenced by the program will appear in the Segment Symbol Table.

An object map will be generated for each block in the source program. This gives the starting location in the text segment of each source statement in the block.

- 2) the compiled program uses data directed input or output. If an empty list is used with a "get data" statement, a full Segment Symbol Table will be generated. Otherwise, a "partial" Segment Symbol Table will be created containing only these variables used in data directed input/output statements.

LOCATION

The Segment Symbol Table for a segment <a> resides in section <a.symbol> along with the binding information. (See MPM Reference Data Section 3.2). The sequences of instructions

```
eapbp <a>| [symbol_table]
eapbp bp|0 "Symbol relocation"
```

or

```
eapbp <*symbol>|0
```

are used to obtain a pointer to the symbol header from which all useful information is available.

ORGANIZATION

The Segment Symbol Table is a list structure consisting of interconnected block and symbol nodes. Each procedure or begin block in the source program has a corresponding block node in the Segment Symbol Table; these block nodes are connected so as to reflect the block structure of the source program. Each block node has a list of symbol nodes emanating from it.

These symbol nodes represent declarations defined immediately internal to the blocks (i.e. internal to the block but not internal to any other block internal to it) and correspond to variables, labels, and entries in the source program.

All of the "pointers" used to connect nodes in the Segment Symbol Table are bit strings that give the distance from the start of the node in which the offset occurs to the node pointed to. All accesses must be of the form:

$$P = \text{addrel}(q, q \rightarrow \text{node.offset})$$

A zero value for an offset means that no offset is present. The use of self relative offsets has two advantages:

- 1) absolute relocation may be used
- 2) the entire Segment Symbol Table may be reached given only a pointer to a single node.

THE BLOCK NODE

The declaration for a block node is:

```
dcl      1  symbol_block      aligned based,
          2  type            unaligned bit (12),
          2  number          unaligned bit (16),
          2  start          unaligned bit (18),
          2  name           unaligned bit (18),
          2  brother        unaligned bit (18),
          2  father        unaligned bit (18),
          2  son            unaligned bit (18),
          2  map            unaligned,
          3  first          bit (18),
          3  last           bit (18),
          2  bits           unaligned bit (18),
          2  header        unaligned bit (18),
          2  chain (4)     unaligned bit (18),
          2  class_list (0:15) unaligned bit (18);
```

"type" is always 516 in a block node,

"number" is not defined at the present time.

"start" is a self relative pointer (SRP) to the symbol node of the first declaration in the block, this declaration list gives all level

0 (non-structure) and level 1 (top level structure) symbols defined immediately internal to the block. This list is ordered according to increasing size in characters of the ASCII name of a symbol and alphabetically within each size group.

"name" is an SRP to an ACC string giving the name of the block. This field will be empty for a begin block. (See MPM Reference Data Section 3.4, Page 2) *in a description of the ACC string representation.*

"brother" is a SRP to the next block node at the same nesting level. It will be 0 if the block does not have a brother.

"father" is a SRP to the immediately containing block node of which the current block is a son.

"son" is a SRP to the first block contained within the current block. It will be 0 if the block does not have a son.

"first" is an SRP to the first word of the object map for this block. Each word in the object map corresponds to a statement in the source program. The left 18 bits of the word contain the starting location in the source program of the statement whose line number is given in the right half.

"last" is an SRP to the last word of the object map for this block.

"bits" are control bits used by the PL/1 compiler.

"header" is an SRP to the symbol header

"chain" is an array of SRPs pointing into the declaration list. $\text{Chain}(i)$ points to the first declaration whose name has 2^{**i} or more characters in it. $\text{Chain}(i)$ will be zero if the largest name in the declaration list is smaller than 2^{**i} .

"class_list" is an array of SRPs pointing into the declaration list. $\text{Class_list}(j)$ points to the first declaration having storage class code equal to j . $\text{Class_list}(j)$ will be zero if no declaration exists in the block having the appropriate storage class.

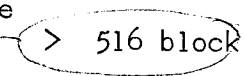
THE SYMBOL NODE

The declaration for a symbol node is:

```
dcl      1  symbol_node      aligned based,
         2  type              unaligned bit (12),
         2  level            unaligned bit (6),
         2  ndims            unaligned bit (6),
         2  bits             unaligned,
           3  aligned        bit (1),
           3  packed        bit (1),
           3  simple        bit (1),
           3  decimal       bit (1),
         2  scale            unaligned bit (8),
         2  name             unaligned bit (18),
         2  brother         unaligned bit (18),
         2  father         unaligned bit (18),
         2  son             unaligned bit (18),
         2  address         unaligned,
           3  offset        bit (18),
           3  class        bit (4),
           3  next         bit (14),
         2  size            fixed bin (35)
         2  word_offset     fixed bin (35)
         2  bit_offset     fixed bin (35)
         2  virtual_org    fixed bin (35)
         2  bounds (ndims)
           3  lower        fixed bin (35)
           3  upper        fixed bin (35)
           3  multiplier   fixed bin (35)
```

"type" is the data type of the item being declared. This uses the same encoding as PL/1 descriptors. The type codes are:

- 1 single precision real fixed point
- 2 double precision real fixed point
- 3 single precision real floating point

4 double precision real floating point
 5 single precision complex fixed point
 6 double precision complex fixed point
 7 single precision complex floating point
 8 double precision complex floating point
 13 pointer data
 14 offset data
 15 label data
 16 entry data
 17-24 arrays of types 1-8
 29-31 arrays of types 13-15
 33 external procedure
 35 internal procedure
 36 entry
 37 label constant
 514 structure
 518 area  516 block
 519 bit string
 520 character string
 521 varying bit string
 522 varying character string
 523-528 arrays of types 514-522

"level" is the structure level number of the declaration. 0 means the declaration is not a structure, and 1 means the declaration is the top level of a structure.

"ndims" is the number of array dimensions for the declaration, 0 meaning the declaration is not an array.

"aligned" is "1"b ^{if} is the variable occupies an entire word.

"packed" is "1"b is the units of the multiplier are bits instead of words.

"simple" is "1"b if the symbol is not an array and has both a zero word offset and a zero bit offset.

"decimal" is "1"b if the arithmetic variable being defined has decimal precision and scale rather than binary.

"scale" is the arithmetic scale factor ranging from -127 to +127.

"name" is a SRP to an ACC string giving the name of the symbol being defined. This string is used to order the declaration list of the block node.

"brother" is an SRP to the next symbol in the declaration list at the same structure level (0 and 1 are considered to be the same level). This field will be zero in the last symbol node on the declaration list.

"father" is a SRP to either a block or a symbol node. If level ≤ 1 father points to the block node in which the symbol is declared. If level > 1 (ie. the symbol is a member of a structure), father points back to the immediately containing symbol node at a structure level one less and which contains the current symbol node as a son.

"son" is a SRP to the first son of a structure, ie. the symbol node for the first variable with a structure level number one greater than the current level.

"offset" is the offset of the datum within its storage class.

"class" is the storage class of the symbol. The following encoding is used:

- 0 unknown
- 1 automatic; "offset" is the offset in the stack frame
- 2 automatic adjustable; the address of the datum is not known at the time the Segment Symbol Table is created. "offset" is the location of the automatic pointer which is used to address the adjustable datum after it is allocated.
- 3 based; "offset" is an SRP to the symbol node for the pointer mentioned in the based declaration. "offset" will be zero if no pointer was specified.
- 4 internal static; "offset" is the offset in ^{the} linkage section.
- 5 external static; a link to the external variable is located at "offset" in the linkage section.
- 6 internal controlled (not available yet)
- 7 external controlled (not available yet)

- 8 parameter; "offset" is the location of the automatic ^{pointer} ptr which is used to address the parameter
- 12 text reference; "offset" is the location of the item in the text segment
- 13 link reference; "offset" is the location of the datum in the linkage section

"next" is a SRP to the next top-level (level \leq 1) symbol having the same storage class.

"size" is an "encoded" representation of the string size or arithmetic precision of the symbol. Since the values may not be known during compilation, procedure "stu_\$\$decode_value" must be called to decide the value. (see below)

If the bit "simple" is on, all of the fields described below are not present. In this case, the word and bit offset of the symbol are both zero.

"word_offset" is the encoded value of the offset of the first word of the symbol from the level 1 containing structure. Procedure stu_\$\$decode_value must be called to determine its value.

"bit_offset" is the encoded value of the offset of the datum in bits from the beginning of the word containing the datum. Procedure stu_\$\$decode_value must be called to determine its value.

The complete sequence for calculating the word and bit offsets of a variable is:

```
wo= stu_$$decode_value (..... )
bo= stu_$$decode_value (..... )
wo= wo + divide (bo,36,17,0):
bo= mod (bo,36);
```

"virtual_org" is the encoded value of the virtual origin of an array datum - its value should be subtracted from the base address obtained from "class" and "offset". Procedure stu_\$\$decode_value must be called to determine the value of the virtual origin.

"bounds (i)" gives the lower and upper array bounds as well as the multiplier for the ith dimension of an array. `stu_decode_value` should be called to obtain the value of any of the bounds information.

The formula for calculating the address of a datum is

$$\text{address} = \text{base_address} - \text{virtual_origin} + \sum_{i=1}^{\text{ndims}} \text{multiplier}(i) * \text{subscript}(i)$$

If the "packed" bit is on, the result of the calculation given above is in bits and should be converted to bits and words; this address should then be combined with the word and bit offsets evaluated earlier.

ENCODED VALUES

Since the size or array bounds of a variable in PL/1 may be adjustable, the following algorithm has been used to encode the rule for obtaining a value only determinable during execution of the program containing the declaration.

The 36 bit word in the symbol node, which was declared as fixed bin (35) actually has the following structure:

dcl	1	encoded_value	aligned based,
	2	flag	unaligned bit (2),
	2	code	unaligned bit (4),
	2	(n1, n2)	unaligned bit (6),
	2	n3	unaligned bit (18);

If `flag="00"b` or `flag="11"b`, the value is the constant value given in the entire word. If `flag="10"b`, the value is encoded according to the following table.

<u>CODE</u>	<u>VALUE</u>
0000	automatic variable at offset n3 in stack frame of the block n1 levels before the block in which the declaration occurs.
0001	internal static variable located at n3 in the linkage section of procedure owning declaration.
0010	external static variable with positive word offset n1 from link located at n3 in the linkage section.

- 0011 value is the bit offset field of the pointer used to qualify a generation of based data plus the additional offset n3.
- 0100 value is based with positive word offset n2 on an automatic pointer located at n3 in the stack frame of the block n1 levels before blocks of procedure owning declaration.
- 0101 value is based with positive word offset n2 on the internal static pointer located at n3 in the linkage section.
- 0110 value is based with positive word offset n2 on the external static pointer with positive word offset n1 located at n3 in the linkage segment.
- 0111 value is based with positive word offset n2 on the pointer used to qualify a generation of based data.
- 1000 value is given by an internal procedure located at n3 in the text segment.

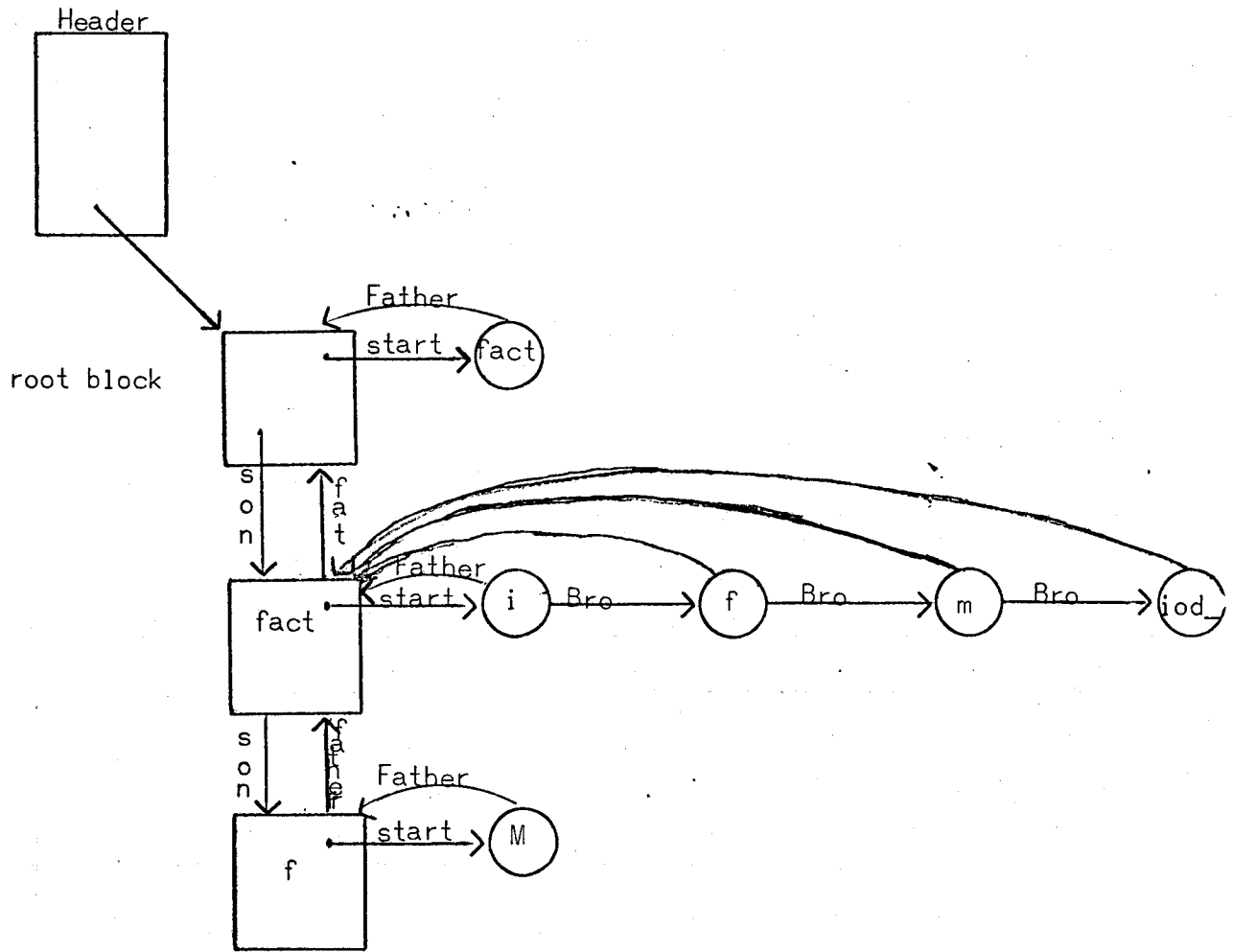
EXAMPLES

We diagram below the overall structure of the symbol table for the procedure given below. Block nodes are shown as squares, and symbol nodes as circles:

```
fact: proc(m);

dcl (i,m) fixed bin,
    f int entry(fixed bin) returns(fixed bin), ioa-entry;
do i = 1 to m;
    call ioa_("7d ^d", i,f(1)); end;
```

```
f:      proc(M) returns(fixed bin);  
  
dcl    M fixed bin;  
      if M = 1 then return(1);  
      else return(M*f(M-1)).  
      end;  
  
end;
```



The structure

```
dcl 1 a
    2 b      fixed,
    2 c
    3 d      cha (4)
    3 e      bit (2)
    2 f      float;
```

would ~~be~~ have the following form of table entry

