TO: MULTICS DISTRIBUTION
FROM: D.B. Wagner
SUBJ: The Segment Symbol Table

The attached revisions of BD.1.00 and BD.1.02 are the final iteration in the design for the Segment Symbol Table. Since programming is in progress on several programs which use the symbol table, no further revisions can be contemplated other than simple extensions.

Changes include:

1. The formats of the "header" and the "nodes" have changed.

2. Much more information is now standard for all translators.

3. Explanations have been made much more clear (I hope).

4. Special in-references to the nodes for entries are included.

Published: 2/17/67
(Supersedes: BD.1.00, 09/07/66;
BD.2.00, 07/12/66)

Identification

Standard Format for the Segment Symbol Table
D.B. Wagner

Purpose

Every translator used in Multics produces as a by-product of every translation a Segment Symbol Table which supplies information about the translation: such information as the storage assigned to variables, their attributes, the version of the translator, the date and time of the translation, and so forth.

There are five major uses of the Segment Symbol Table in Multics:

1. The debugging aids (see BX.10) need the information in the symbol table in order to be able to answer a user's questions about the workings of a program when the questions are asked in terms of the source language.

2. When the Shell calls a command procedure it needs information on the attributes of the arguments expected by the command. It gets this information from the Segment Symbol Table.

3. When a call crosses a protection wall (see BD.9.02) outward, a Gatekeeper must copy the call's arguments into a data segment which is accessible in the called program's ring. In order to copy these arguments the Gatekeeper needs information on their attributes. To provide this information all calls which might cross outward have pointers into a Segment Symbol Table attached to the argument list. See BD.7.02 for details.

4. The call-passer (described in BD.8.02) provides the ability to make procedure calls across processes. Since segment numbers are process-dependent, arguments in these calls must be converted to a standard form (using "slot numbers"). Here the same mechanism of symbol-table pointers tacked onto the argument list must be used so that the call_passer can know how to process the arguments.

5. Implementation of "data-directed input" will require a symbol table. See the PL/I manual, IBM form C28-6571-3, pp. 90 - 92 for PL/I's version of data-directed input.

The design for the Segment Symbol Table presented here makes an attempt at the difficult task of solidifying those parts of the table which are essential to system performance but at the same time allowing flexibility in the design of translators.

Definitions

An identifier is as in PL/I: a string of characters used as an indivisible entity in a source program. A symbol is an identifier defined by a programmer in a source program (i.e., an identifier whose use is not fixed in the language). In PL/I for example a symbol may be a scalar variable name, structure name, array name, statement label (including block name and entry name), external entry name, segment name, or condition name.

A segment symbol table is that by-product of a compilation or assembly which provides information about the symbols used in the associated program segment. It contains, for each symbol defined in the source program, two kinds of information: implementation information and attribute information. Implementation information provides the necessary correlation between the source program and the associated object program. It includes the addresses of variables. Attribute information is information present in the source program such attributes of variables.

Location of Symbol Table

The Segment Symbol Table for a segment < a > normally resides in the segment < a.symbol > along with binding information (see BD.2.01). On the whim of the translator, however, it may reside in < a > or even < a.link >. It is always accessed through the reference

< a >|[symbol_table]

and information in a's linkage section tells the linker where the symbol table actually is.

In order to make information on entries more accessible to programs like the Shell, special in-references should be added to a procedure's linkage section pointing to the symbol-table

nodes for all that procedure's entries. For an entry

< a >|[b]

the symbol table node should be at

< a >|[b_smtb_]

The suffix "_smtb_" must be registered in the Name Registry (section BB.).

The Symbol Table

The symbol table is structured, that is in the form of a tree, for at least two reasons: First, PL/I blocks may be nested. Second, the best way to give information concerning a PL/I data-structure is in a tree. There are also other places where structuring is useful. For example, the entry in the symbol table for a PL/I pointer variable may contain pointers to the entries for any based variables which have this pointer specified in their declarations. Languages other than PL/I will undoubtedly have other uses for structuring in the symbol table.

All structuring of the table is done in a standard way which does not depend on content. This is done so that a standard subroutine can be used to interpret the structure, no matter what translator produced the table or what the structuring "actually means". At < a >|[symbol_table] is the header for the table. This is a collection of information concerning the translation such as the translator name, the date and time of the translation, the name of the source file, etc.

The tree-structuring is accomplished using what are called symbol table nodes. A symbol table node is a collection of self-relative pointers pointing to other nodes, plus a self-relative pointer to an information block giving information about the symbol (if any) to which the node corresponds.

The information block is an agglomeration of bit-strings giving attributes, addresses, etc., for the symbol, and also of course the name of the symbol. Its design is almost entirely up to the designer of the translator, with some simple restrictions noted here.

Included in the header of the symbol table is the information block map which brings some order out of the chaos of the information block. It includes enough information so that as long as use is restricted to the commonest data types (e.g. PL/I scalars) the information block can be accessed without reference to what translator produced the table. There will always be some lag in updating debugging aids, etc., to work with new translators, and the information block map allows these programs to give some help to the user until such updating has been performed. It also allows a gatekeeper to be fairly efficient.

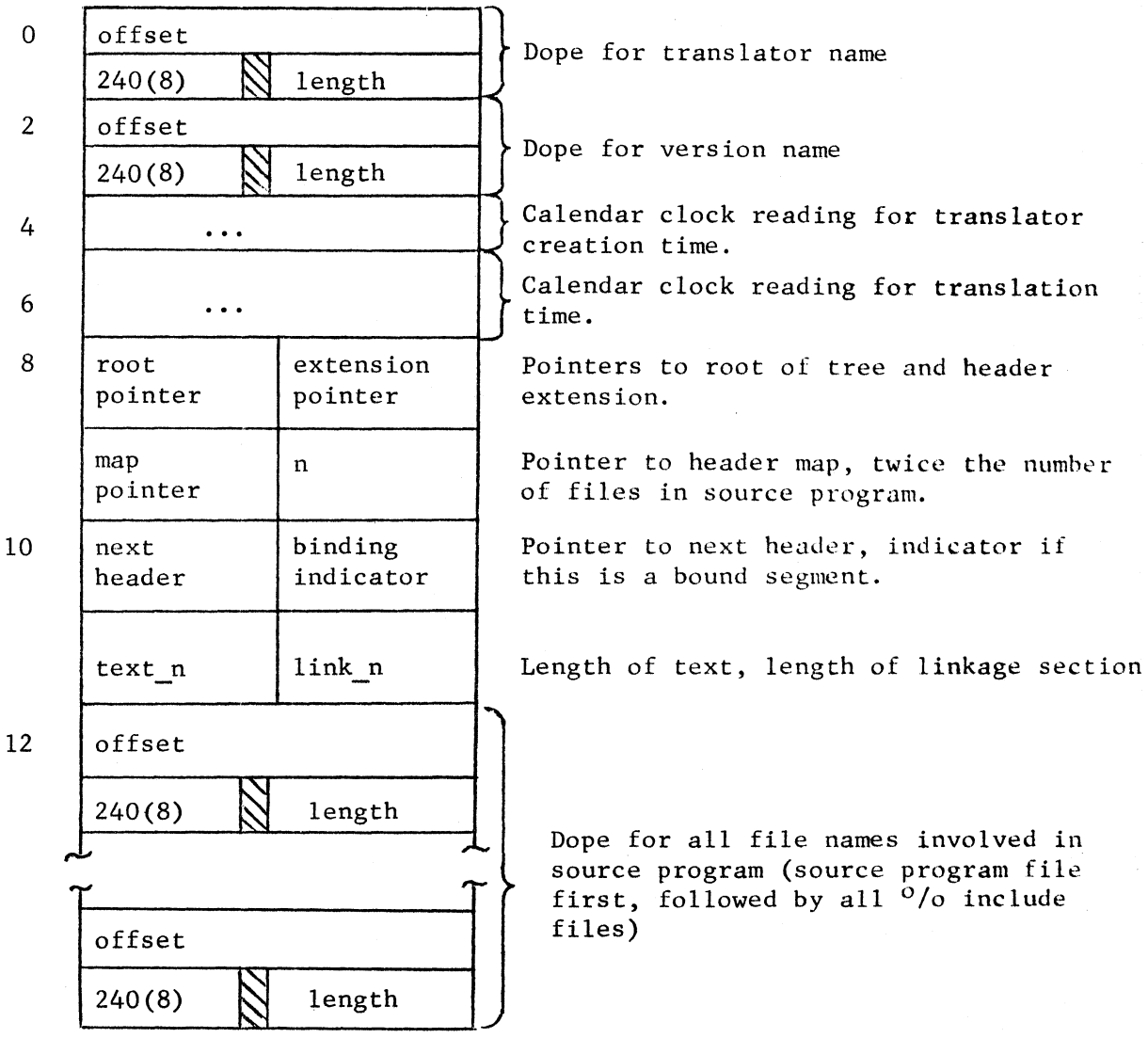
Published: 06/19/68

Identification

Revision of BD.1.00
R. M. Graham

Revisions

The following figure of the header should be substituted for the one on page 5.



The various names shown in the diagram above are indicated using PL/I - style string dope. See BP.2.02 for details. The "offset" is the offset in bits from word 0 of the header, and the "length" is the length in bits of the string.

The "translator name" is something like "pl1" or "eplbsa" - normally it is the name of the translator command. The debugging aids' expression-evaluation routines will use the translator-name with various suffixes to form names of procedures (for example to handle the translator's data-types) so that this name should include only letters, digits, and the underscore.

The "translator version name" is a short printable line such as "epl version 6 level 0" or "J.G.'s special test version with optimized structure accessing".

The "root pointer" is a self-relative pointer to the root node of the tree. See Overall Structure, below.

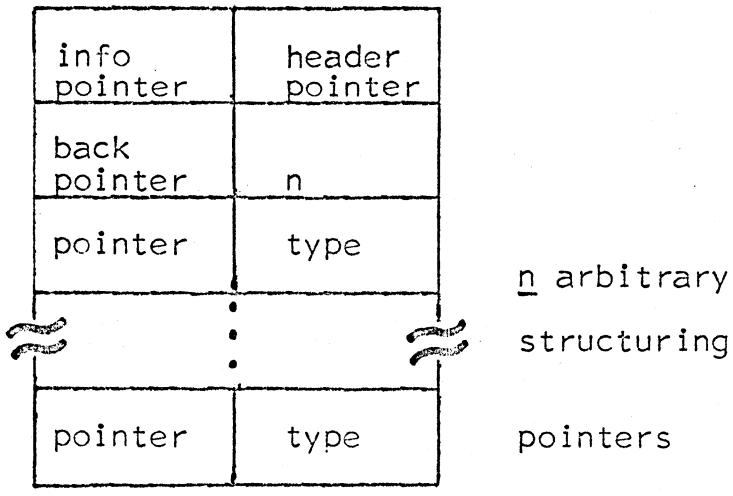
The "map pointer" is a self-relative pointer to the information block map. The information block map is described below.

"n" is twice the number of files involved in the source program. This may be greater than 2 if, for example, the include feature of PL/I is used.

"Next header" and "binding indicator" are non-zero only when segments are bound together. See BD.14.01 for details. Roughly, when segments are bound together the headers for the separate symbol tables are chained together using the self-relative pointer "next header", and "binding indicator" is non-zero in each of these headers.

The Node

A symbol table node is as follows:



All the indicated "pointers" are self-relative. A zero pointer is considered null.

"Info pointer" points to an information block.

"Header pointer" points to the header of the symbol table. Therefore given a pointer to a symbol table node a program can always locate the information block map.

"Back pointer" points to this node's immediate parent node, or else is null.

"n" is the number of "structuring pointers" which follow. Each structuring pointer points to another symbol table node somewhere else in the same segment.

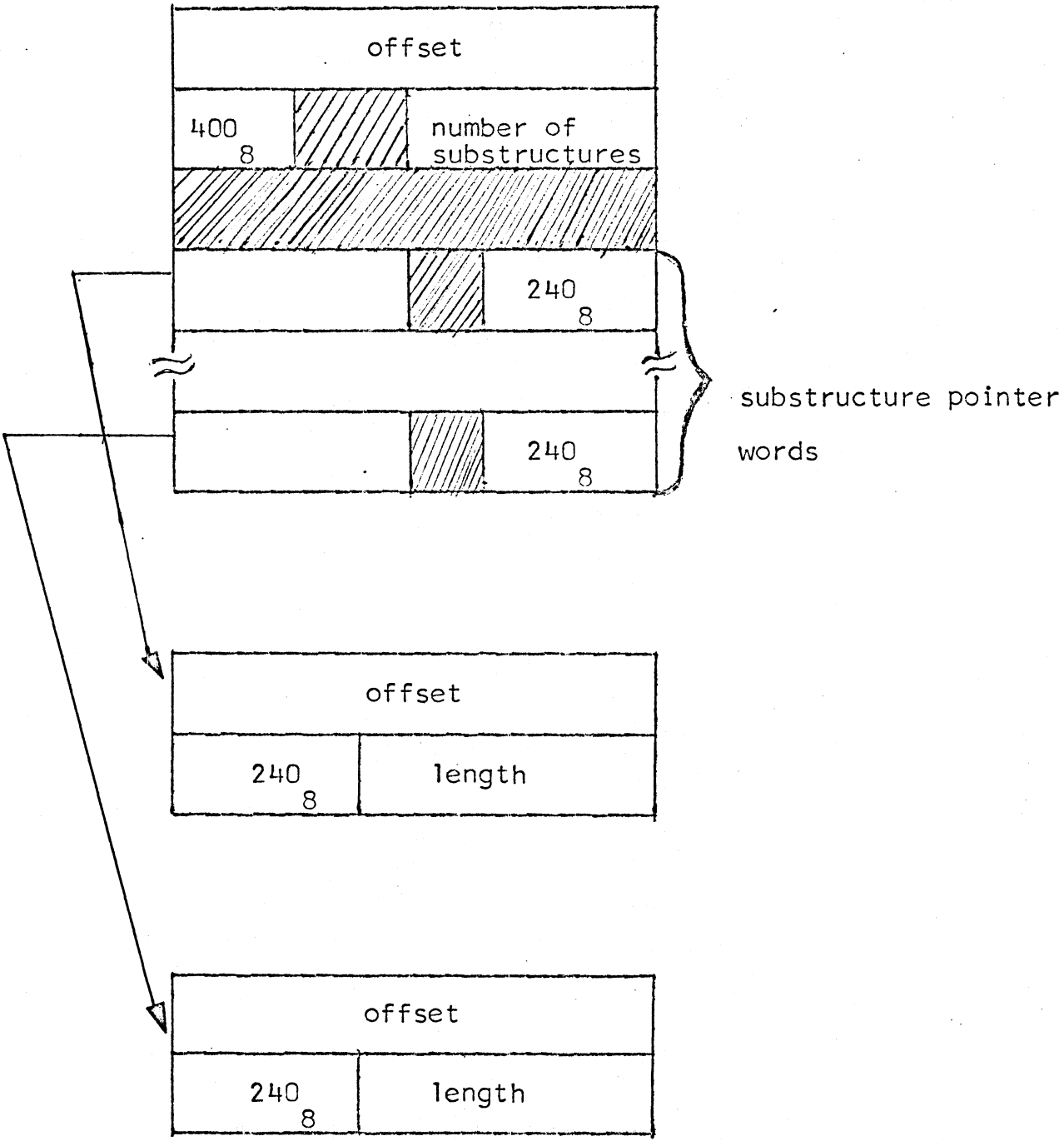
Each structuring pointer in the node may have associated with it an 18-bit "type number" particular to the translator involved. In the PL/I symbol table, for example, type numbers are defined for the pointer-variable controlling a based variable, pointers to substructures of a data-structure, and other kinds of pointers.

The structuring pointers may be considered to be of two kinds: "branches" and "links" (using the terminology made popular by the file system). Branches are pointers to nodes which logically are "descendants" of the present node in the tree structure and links are pointers to nodes which are best considered to be "cutting across" the tree structure. Within the symbol table a branch is defined to be a pointer to a node whose back-pointer points to the present node, and a link is defined to be a pointer to a node whose back-pointer points to some other node (or is null, as in the case of the root of the tree). This distinction is important for the following reason: a symbol-table searching procedure can search the tree and be absolutely secure against loops simply by ignoring links (see, for example, BY.6.02).

The Information Block and its Map

The information block contains information about a symbol such as its name, symbol type, address or addresses, precision, and whatever information the translator possesses and the translator designer feels is worth putting into the symbol table.

The information block map found in the symbol table header looks like a PL/I dope vector for a structure of strings (see BP.2.02):



Each of the "substructure pointer words" points to string dope for one item (e.g. name, data type, precision, etc.). The "offset" in the string dope gives the offset (in bits) of the item in the information block, and "length" gives the length in bits of the item. The items may be arranged in the information block according to the whim of the translator designer: in particular items may overlap when this turns out to be a reasonable way of doing things. Translator documentation must tell what items appear in the information block and in what order they appear in the map.

In the case of variable-length items (such as the symbol name) a separate item in the information block can be used to give the length. Then the string dope for the variable-length item will contain a meaningless "length".

Similarly it may be worthwhile to let some item's offset in the information block be specified by another item. This variable offset does not seem terribly useful however.

Standard Items in the Information Block

The information block for a symbol may contain any information which seems reasonable, but certain items are standard and must be in fixed places in the information block map:

1. symbol type
2. address
3. address type
4. length of name in bits
5. name

"Symbol type" is a number telling what the symbol is used for in the program. The following symbol type numbers are standard:

0 = unknown

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 = varying bit-string

11 = non-varying character-string

12 = varying character-string

13 = pointer-variable (its pair)

14 = relative pointer-variable (offset from another pointer)

15 = label variable

16 = entry variable

17-32 = array of one of the above

33 = external procedure

34 = external data region

35 = internal procedure

36 = entry

37 = label constant

Most cases above where a symbol-type is named refer to the PL/I implementation of the same. See BP.2.01 and BP.2.02 for details. PL/I does not have a data-type "entry variable"; it does however allow "entry parameters" and it is to the implementation of these that we refer. "External procedure" and "external data region" are special concepts described later (see Overall Structure).

The above list includes all of the symbol-types which may be passed in an outward ring-crossing. They are of course also the ones which will be most common in the various languages seen in Multics.

Symbol-type numbers below 512 are reserved for coordinated expansion of the above list. Numbers above 512 may be used for peculiar symbol-types germane to only one translator.

"Address type" is a number specifying how "address" is to be interpreted:

0 = irrelevant (no meaningful address)

1 = location in segment

2 = argument number

3 = stack address (offset from sb◁—sp)

4 = linkage address (offset from lb◁—lp of indirect word)

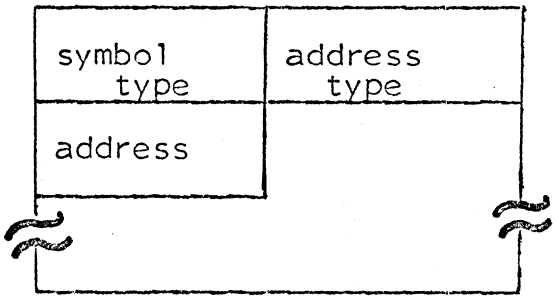
5 = address in symbol table (offset from <segname>|[symbol_table])

6 = address in the segment <segname.symbol>

7 = address in the segment <segname.link>(for entries).

Address-type's up to 512 are reserved for coordinated expansion of this list. Numbers above 512 may be used for peculiar address-interpretations germane to only one translator.

For the sake of efficiency in the gatekeeper and the call-passer, the first three of these items must be placed in standard positions in the information block itself. The required format is:



Overall Structure

This much of the overall structure is standard: "root pointer" in the symbol table header points to a node whose associated information block has symbol-type equal to either external procedure or external data region. If it is external procedure then for each external entry into the segment there is a structuring pointer pointing to a node whose associated information block includes the entry name and has symbol-type = entry. The node for the entry contains structuring pointers for the arguments expected at the entry point. Each of the information blocks for arguments to the entry has "address-type"= argument number.

If symbol-type = external data region (indicating that the segment is a data segment accessed through in-references in its linkage section) then for each in-reference in the segment's linkage section there is a structuring pointer pointing to a node whose associated information block gives the data-type etc., of the data to be found through that in-reference.

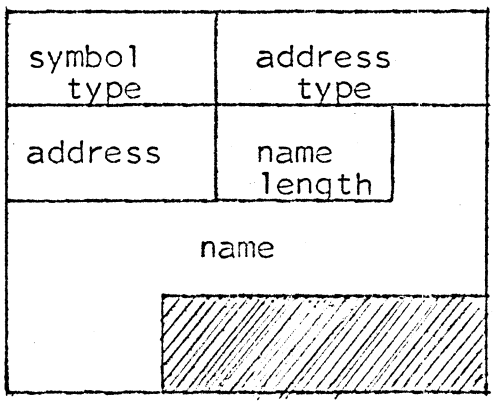
These nodes will of course contain all manner of other structuring pointers peculiar to the translator as well as those legislated above. The required nodes can be distinguished from the others by the symbol-types in the associated information blocks.

Example

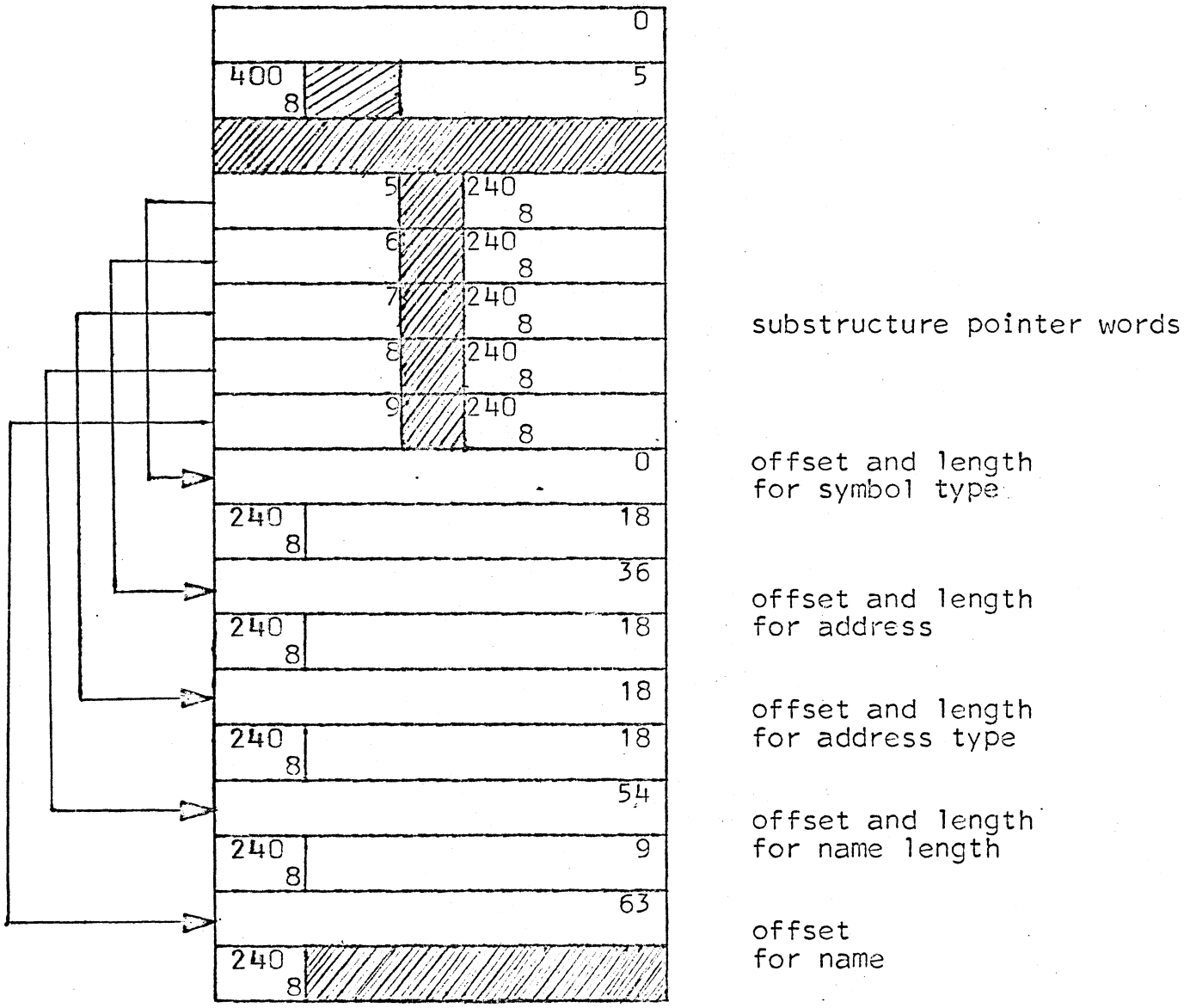
Presented here as an example is a possible symbol-table format for a MAD translator used in Multics.

All of MAD's symbol-types and address-types are in the standard lists above. The block-structuring permitted is quite simple. Further, no information except that which is listed above as "required" is kept by the translator. For these reasons the symbol table can be quite simple.

The information block has the following format:



Thus the information block map looks as follows (compare with the diagram given earlier).



The following structuring is added to the standard structuring described above. Every internal procedure has a node branching

from the external procedure node (in MAD, internal procedures may not be nested), and every variable used in the program has a node branching from the external procedure.

Thus the program:

```
EXTERNAL FUNCTION (ALPHA, BETA, GAMMA)
ENTRY TO Q1.
DIMENSION Z (100)
INTEGER ALPHA
...
INTERNAL FUNCTION
ENTRY TO QQ1.
Y = Z (7)
END OF FUNCTION
END OF FUNCTION
```

Would have a symbol table which might be diagrammed as shown in Figure 1.

Particular Symbol Tables.

The symbol table produced by any particular translator must be documented in an MSPM section numbered BD.1.XX. This documentation should include a list of all uses of structuring and the type numbers associated with the pointers in the nodes, a description of each item in the information block, and a description of each special address-type and symbol-type used.

In the design of the symbol-table for a particular translator, some points should be borne in mind. (1) If an existing symbol-table format can be used, it should be (for example the Fortran IV symbol table can probably have the same format as the PL/I symbol table). This will make the writing of the programs which use the symbol table much easier. If no existing format can be used, an extension of an existing one may be possible. (2)Within reason, the symbol-table should include all the information which the translator happens to have in its own internal tables concerning the use and implementation of a symbol, even if no use can be seen for the information. There is no telling what peculiar kinds of debugging aids may become useful for dealing with higher-language programs. (3) If more than one address is to be included in the information block (for example in PL/I the addresses of the specifier, dope, data, and free-storage area of a varying string), the one which seems logically most basic

should be given the honored first position in the map (in PL/I this would be the address of the specifier if there is one, and the address of the data if there is not).

If the symbol-table format for a translator is extended at any time, the information-block may be reorganized and the map changed accordingly, with all additions to the information block added to the end of the map regardless of where they go in the block. Thus many changes in the translator need not be tightly coordinated with changes in the programs which use its symbol-table.

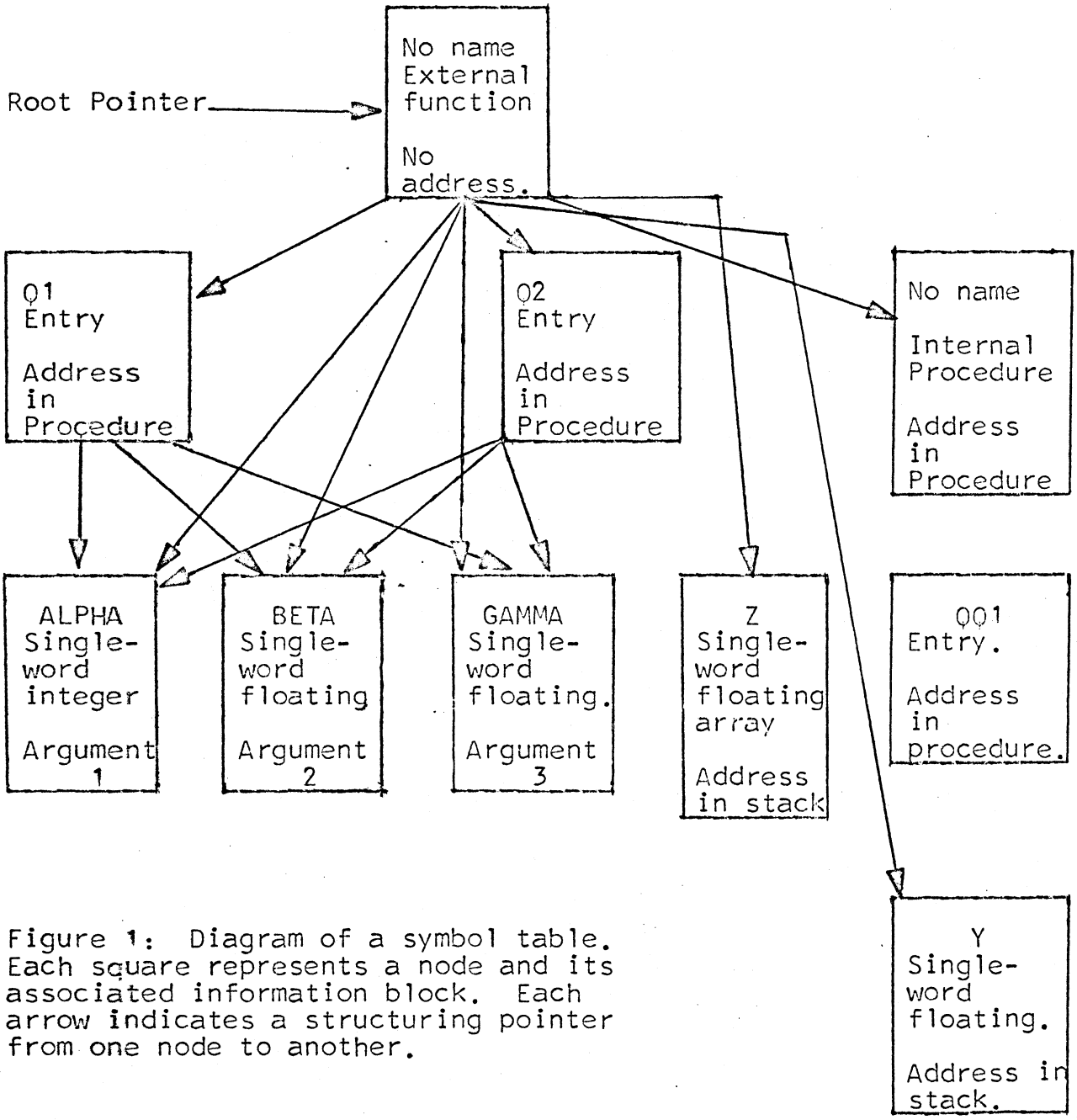


Figure 1: Diagram of a symbol table. Each square represents a node and its associated information block. Each arrow indicates a structuring pointer from one node to another.