MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

The Multiplexed Information and Computing Service: Programmers' Manual

PART III
SUBSYSTEM WRITERS' GUIDE TO MULTICS

Revision 1

Date: 5/31/73

PREFACE

PREFACE TO THE MULTICS SUBSYSTEM WRITERS' GUIDE

January 10, 1973

This volume of the MPM has as its purpose to provide constructors of sophisticated Multics subsystems the extra information needed for their undertaking. The Multics program modules documented here represent a layer of penetration into the system beyond that which is intended for the casual user. As a result, there are several caveats in order:

1) The interfaces described here are not generally as neatly organized or consistently thought out as is the standard user interface represented by the MPM Reference Guide because they are in an earlier stage of development.

2) Most of the interfaces provided in this manual would be used only if there is some reason to bypass the usual standard system way of doing things. As a result, in using an SWG interface, one risks giving up some features of Multics. Although there is no claim that all the features of Multics are essential, there is a consistency of style and interpretation at the standard interface which the ultimate subsystem user may be accustomed to. The subsystem writer should be cautious about unintentionally introducing different styles and interpretations when bypassing a standard function.

3) All of the facilities described in the SWS are subject to changes and improvements in their interface specifications. Further, at the level of penetration represented by the SWG interfaces, it is difficult to avoid large disruptions when interfaces are changed. Thus, the subsystem writer is strongly cautioned against unnecessarily utilizing the SWG interfaces.

4) The initial release of the manual consists in part of simple reproduction of sections formerly found in the Subsystem Writers' Supplement to the MPM. Many of these sections should be expanded and reformatted, especially where they consist of nothing but a calling sequence and notes. Thus,

for example, many sections do not include adequate warnings about the implications of using the module being described.

5) The information about the Multics environment necessary to intelligently use the SWG interfaces is frequently not available in any form. This unfortunate situation should be slowly remedied through accumulated updates to the SWG.

Thus it should be clear that the subsystem writer utilizes the SWG interfaces at his own risk; and with knowledge of the pitfalls involved. On the other hand, one of the primary objectives of Multics is to allow subsystems of almost any arbitrary specification to be constructed. To that end, the SWG interfaces are intended to be helpful.

 (END)

FOREWORD

PLAN OF THE MULTICS PROGRAMMERS' MANUAL

January 10, 1973

The Multics Programmers' Manual (MPM) is the primary reference manual for user and subsystem programming on the Multics system. It is divided into three major parts:

Part I: Introduction to Multics

Part II: Reference Guide to Multics

Part III: Subsystem Writers' Guide to Multics

Part I is an introduction to the properties, concepts, and usage of the Multics system. Its four chapters are designed for reading continuity rather than for reference or completeness. Chapter 1 provides a broad overview. Chapter 2 goes into the concepts underlying Multics. Chapter 3 is a tutorial guide to the mechanics of using the system, with illustrative examples of terminal sessions. Chapter 4 provides a series of examples of programming in the Multics environment.

Part II is a self-contained comprehensive reference guide to the use of the Multics system for most users. In contrast to Part I, the Reference Guide is intended to document every detail and to permit rapid location of desired information, rather than to facilitate cover-to-cover reading.

Part II is organized into ten sections, of which the first eight systematically document the overall mechanics, conventions, and usage of the system. The last two sections of the Reference Guide are alphabetically organized lists of standard Multics commands and subroutines, respectively, giving details of the calling sequence and the usage of each.

Several cross-reference facilities help locate information in the Reference Guide:

- The table of contents, at the front of the manual, provides the name of each section and subsection and an alphabetically ordered list of command and subroutine names.

- A comprehensive index (of Part II only) lists items by subject.

- Reference Guide sections 1.1 and 2.1 provide lists of commands and subroutines, respectively, by functional category.

Part III is a reference guide for subsystem writers. It is of interest to compiler writers and writers of sophisticated subsystems. It documents user-accessible modules which allow a user to bypass standard Multics facilities. The interfaces thus documented are a level deeper into the system than those required by the casual user.

Examples of specialized subsystems for which construction would require reference to Part III are:

1) a subsystem which precisely imitates the command environment of some system other than Multics (e.g., an imitation of the Dartmouth Time-Sharing System);

2) a subsystem which is intended to enforce restrictions on the services available to a set of users (e.g., an APL-only subsystem for use in an academic class);

3) a subsystem which is protecting some kind of information in a way not easily expressible with ordinary access control lists (e.g., a proprietary linear programming system, or an administrative data base system which permits access only to program-defined aggregated information such as averages and correlations).

Each of the three parts of the MPM has its own table of contents and is updated separately, by adding and replacing individual sections. Each section is separately dated, both on the section itself, and in the appropriate table of contents. The title page and table of contents are replaced as part of each update, so one can quickly determine if his manual is properly up-to-date. The Multics on-line "message of the day" or local installation bulletins should provide notice of availability of new updates. In addition, the Multics command "help mpm" provides on-line information about known errors and the latest MPM update level.

In addition to this manual, users who will write programs for Multics will need a manual giving specific details of the language they will use; such manuals are currently available for PL/I, FORTRAN, and BASIC. A separate, specialized supplement to the MPM is also provided for users of graphic displays. The bibliography at the end of Part I, Chapter 1, describes these and other references in more detail.

Multics provides the ability for a local installation to develop an installation-maintained or author-maintained library of commands and subroutines which are tailored to local needs. The installation may also document these facilities in the same format as used in the MPM; the user can then interfile these locally provided write-ups in the command and subroutine sections of his MPM.

Finally, access to Multics requires authorization. The prospective user must negotiate with the administration of his local installation for permission to use the system. The installation may find it useful to provide the new user with a documentation kit describing available documents, telephone numbers, operational schedules, consulting services, and other local conventions.

CONTENTS

May 31, 1973

Section 16 Subroutines (1/10/73)

continued on next page

Section 16 Subroutines (continued)

 (END)

STANDARD OBJECT SEGMENT: INTRODUCTION AND OVERVIEW

A Multics object segment is a segment which contains named items which are externally accessible via the dynamic linking mechanism. (See the MPM Subsystem Writers' Guide section, Dynamic Linking.) The most common examples of object segments are procedure segments and data segments. (The segment error_table_ is an example of the latter).

The following documents describe the format of the standard Multics object segment. The format requirements relate primarily to the external interfaces of an object segment, giving the translator great freedom in the area of code and data generation. The format contains certain redundancies and some rather odd data structures. This is due to the necessity for maintaining upwards compatibility with earlier object segment formats. The dynamic linking mechanism and the standard object segment manipulation tools assume that all object segments are standard object segments.

The information in an object segment falls into several categories (not all of which necessarily appear in a given object segment):

1) Text	pure (nonselfmodifying) machine instructions and/or pure data.
2) Definitions	symbolic information with the aid of which certain variables which are internal to the object segment are made known to the external world and accessible to the dynamic linking mechanism (the linker).
3) Links	symbolic representation of names whose addresses are unknown at compile time, and can only be evaluated (i.e., resolved into machine addresses by the linker) at execution time.
4) Symbol Tree	internal representation of symbolic source language variables, their attributes and addresses; needed for the execution of interpretive code such as PL/I data-directed input/output as well as for debugging purposes.

5) Historical Information — information describing the circumstances under which the object segment was created, such as the name and version of the translator, creation time, identification of input source, identification of user who initiated the object segment creation, etc.

6) Relocation Information — information which identifies all instances of internal relative address references to enable their relocation.

7) Diagnostic Aids — information which allows standard system tools to extract useful information out of an object segment.

The information items listed above are not stored, intermixed, within a monolithic object segment. Rather, the object segment is divided into five sections named text, definition, linkage, symbol and break map. The object segment is a concatenation of these five sections in the following order:

text

definition

linkage

symbol

break map (if present)

The object segment also contains an object map which contains the offsets and lengths of each of the sections. The object map may be located immediately before or immediately after any of the five sections. Translators normally place it immediately after the symbol section. The last word of the segment must contain a left-justified 18-bit relative pointer to the object map. The assignment of any item to one of the five sections is decided as follows:

1) Text Section — contains only the pure parts of the object segment; that is, instructions and read-only data. It may also contain relative pointers into the definition, linkage and symbol sections.

2) Definition Section — contains only nonexecutable read-only symbolic information which is intended for the purposes of dynamic linking and symbolic debugging. It is assumed that the definition section will be infrequently referenced (as opposed to the constantly referenced text section); this section is therefore not recommended as a repository for read-only constants which are referenced during the execution of the text section. The definition section may sometimes (as in the case of an object segment generated by the binder) be structured into definition blocks, which are threaded together.

3) Linkage Section — contains the impure (i.e., modified during the program's execution) nonexecutable parts of the object segment and consists of two types of data:

a) links which are modified at run time by the Multics linker to contain the machine address of external names;

b) data items which are to be allocated on a per-process basis. This includes the internal static storage of PL/I procedures.

4) Break Map Section — contains information which allows the debugger to locate break points in the object segment. This section is generated by the debugger rather than the translator and will not exist unless the segment currently contains break points. Its internal format is of interest only to the debugger and is not described in the MPM.

5) Symbol Section — so named because it was initially designed to store the language processor's symbol tree, is the repository of all generated items of information which do not belong in the

first four sections. The symbol section may typically be further structured into variable length symbol blocks, stored contiguously and threaded to form a list. The symbol section may contain pure information only.

The text, definition and symbol sections are shared by all processes which reference an object segment. When an object segment is first referenced in a process, a copy of the linkage section is made. That is, the linkage section is a per-process data base. The original linkage section serves only as a copying template. See the MPM Subsystem Writers' Guide sections, Dynamic Linking, and Standard Stack and Linkage Area Formats. Normally, a segment with a nonempty break map section is in the process of being debugged and is not used by more than one process.

THE STRUCTURE OF THE TEXT SECTION

The text section is basically unstructured, containing the machine language representation of some symbolic language algorithm and/or pure data items. Its length must be an even number of words.

Two items which may appear within the text section have standard formats; namely the entry sequence and the gate segment entry point transfer vector.

The Entry Sequence

There must be a standard entry sequence for every externally accessible procedure entry point in an object segment. It has the following format.

```
declare 1 entry_sequence aligned,
          2 def_relp bit(18) unaligned,
          2 pad bit(18) unaligned,
          2 code_sequence(n) bit(36) aligned;
```

1) def_relp — is a pointer (relative to the base of the definition section) to the definition (see below) of this entrypoint. Thus, given a pointer to an entrypoint, it is possible to reconstruct its symbolic name for purposes such as diagnostics or debugging.

2) pad — is reserved for future use and must be "0"b.

3) code_sequence — is any sequence of machine instructions satisfying Multics standard calling conventions. See the MPM Subsystem Writers' Guide section, Subroutine Calling Sequences.

Note that the value (i.e., offset within the text section) of the entry point corresponds to the address of the code_sequence item. (The value is stored in the formal definition of the entry point. See the MPM Subsystem Writers' Guide section, The Structure of the Definition Section.) Thus, if entry_offset is the value of the entry point ent1 then the def_relp item pointing to the definition for ent1 is located at word (entry_offset - 1).

The Gate Segment Entry Point Transfer Vector

For reasons of protection, control must not be passed to a gate procedure at other than its defined entry points. To enforce this restriction, the first n words of a gate segment

with n entry points must be an entry point transfer vector. That is, the kth word. ($0 \leq k \leq n-1$) must be a transfer instruction to the kth entry point (i.e. a transfer to the code_sequence item of a standard entry sequence as described above). In this case, the value of the kth entry point is the offset of the kth transfer instruction (i.e. word k of the segment) rather than the offset of the code_sequence item of the kth entry point.

To ensure that only these entries may be used, the hardware enforced call limiter of the gate segment must be set so that the segment may be entered only at the first n locations.

 (END)

THE STRUCTURE OF THE DEFINITION SECTION

The definition section of an object segment contains pure information to be used by the dynamic linking mechanism. Its length must be an even number of words. It must start at an even offset in the object segment.

The definition section consists of a header pointing to a linked list of items describing the externally accessible named items of the object segment, followed by an unstructured area containing information describing the externally accessible named items of other object segments referenced by this object segment. The linked list is known as the definition list. The items on the list are known as definitions. The unstructured area contains expression words, type pairs, trap pairs, trap procedure information, and the symbolic names associated with external references.

A definition specifies the name of an externally accessible named item and its location in the object segment. The definition list consists of one or more definition blocks each of which consists of one or more class-3 definitions followed by zero or more non-class-3 definitions. Normally, unbound object segments contain one definition block whereas bound segments contain one definition block for every externally accessible component object segment.

The information in the unstructured area of the definition section is used in conjunction with information in the linkage section to resolve at run-time the external references made by the object segment. This information may, in fact, be viewed as part of the linkage section. It is stored in the definition section to allow it to be shared among all the users of the segment.

See figure 1 for a diagram of the structure of the definition section. For more information concerning the interpretation of the information in the definition section see the MPM Subsystem Writers' Guide section, Dynamic Linking.

For historical reasons, character strings in the definition section are stored in ALM "acc" format, which may be defined by the following PL/I declaration:

```
declare 1 acc aligned,
          2 length_of_string fixed bin(8) unaligned,
          2 string char(length_of_string) unaligned;
```

The first nine bits of the string contain the length of the string. Such a structure will be referred to as an acc string.

Descriptions of the formats of the various items in the definition section follow.

The Definition Section Header

The definition section header resides at the base of the definition section and contains a pointer (relative to the base of the definition section) to the beginning of the definition list.

```
declare 1 def_header aligned,
          2 def_list_relp bit(18) unaligned,
          2 unused bit(36) unaligned;
          2 flags unaligned,
            3 new_format bit(1) unaligned initial ("1"b),
            3 ignore bit(1) unaligned initial ("1"b),
            3 unused bit(16) unaligned;
```

1) def_list_relp — is a relative pointer to the first definition in the definition list.

2) unused — is reserved for future use and must be "0"b.

3) flags — contains 18 binary indicators to provide information about this definition section:

   new_format — if equal to "1"b, the definition section has the format described in this document as distinct from an older format.

   ignore — if equal to "1"b, and new_format is also equal to "1"b, then the Multics linker recognizes this structure as a definition section header and ignores it.

   unused — is reserved f0r future use and must be "0"b.

The Definition

The format of a non-class-3 definition is as follows:

```
declare 1 definition aligned,
          2 forward_thread bit(18) unaligned,
          2 backward_thread bit(18) unaligned,
          2 value bit(18) unaligned,
```

```
2 flags unaligned,
  3 new_format bit(1) unaligned,
  3 ignore bit(1) unaligned,
  3 entrypoint bit(1) unaligned,
  3 retain bit(1) unaligned,
  3 descr_sw bit(1) unaligned,
  3 unused bit(10) unaligned,
2 class bit(3) unaligned,
2 symbol_relp bit(18) unaligned,
2 segname_relp bit(18) unaligned,
2 n_args bit(18) unaligned,
2 descriptor (n_args) bit(18) unaligned;
```

1) forward_thread — is a thread (relative to the base of the definition section) to the next definition. The thread terminates when it points to a word which is zero. This thread provides a single sequential list of all the definitions within the definition section.

2) backward_thread — is a thread (relative to the base of the definition section) to the preceding definition. The thread terminates when it points to a word which is zero. This thread provides a single sequential list of all the defintions within the definition section.

3) value — is the offset, within the section designated by the class variable (see 5 below), of this symbolic definition.

4) flags — contains 15 binary indicators to provide additional information about this defintion:

   new_format — if equal to "1"b, the definition has the format described in this document, as distinct from an older definition format.

   ignore — if equal to "1"b, the definition does not represent an external symbol and must therefore be ignored by the Multics linker.

   entrypoint — if equal to "1"b, this is the definition of an entrypoint (i.e., a variable referenced through a transfer of control instruction).

retain — if equal to "1"b, the definition must not be deleted from the object segment.

descr_sw — if equal to "1"b, the definition includes an array of argument descriptors (i.e., items n_args and descriptor (n_args) below contain valid information).

unused — is reserved for future use and must be "0"b.

5) class — this field contains a code which indicates which section of the object segment the value is relative to, as follows:

0 — text section

1 — linkage section

2 — symbol section

3 — this symbol is a segment name

6) symbol_relp — is a pointer (relative to the base of the definition section) to an aligned acc string representing the definition's symbolic name.

7) segname_relp — is a pointer (relative to the base of the definition section) to the first class-3 definition of this definition block.

8) n_args — is the number of arguments expected by this external entrypoint. This item is valid only if descr_sw = "1"b.

9) descriptor — is an array of pointers (relative to the base of the text section) which point to the descriptors of the corresponding entrypoint arguments. This item is valid only if descr_sw = "1"b.

In the case of a class-3 definition, the above structure is interpreted as follows:

```
declare 1 segname aligned,
          2 forward_thread bit(18) unaligned,
          2 backward_thread bit(18) unaligned,
          2 segname_thread bit(18) unaligned,
          2 flags bit(15) unaligned,
```

```
        2 class bit(3) unaligned,
        2 symbol_relp bit(18) unaligned,
        2 first_relp bit(18) unaligned;
```

1) forward_thread is as above.

2) backward_thread is as above.

3) segname_thread is a thread (relative to the base of the definition section) to the next class-3 defintion. The thread terminates when it points to a word which is all zero. This thread provides a single sequential list of all class-3 definitions in the object segment.

4) flags is as above.

5) class is as above.

6) symbol_relp is as above.

7) first_relp is a pointer (relative to the base of the definition section) to the first non-class-3 definition of the definition block. If the block contains no non-class-3 definitions it points to the first class-3 definition of the next block. If there is no next block, it points to a word of zeros.

The end of a definition block is determined by one of the following conditions (whichever comes first):

a) forward_thread points to an all zero word;

b) the current entry's class is not 3, and forward_thread points to a class-3 definition;

c) the current definition is class-3, and both forward_thread and first_relp point to the same class-3 definition.

Figure 1 illustrates the threading of definition entries.

The rest of this document deals with items in the unstructured portion of the definition section.

The Expression Word

The expression word is the item pointed to by the expression pointer of an unsnapped link (see the MPM Subsystem Writers' Guide section, The Structure of the Linkage Section), and has the following structure:

```
declare 1 exp_word aligned,
          2 type_pair_relp bit(18) unaligned,
          2 expression bit(18) unaligned;
```

1) type_pair_relp — is a pointer (relative to the base of the definition section) to the link's type-pair.

2) expression — is a signed fixed bin(17) value to be added to the value (i.e., offset within a segment) of the resolved link.

The Type Pair

The type pair is a structure which defines the external symbol pointed to by a link.

```
declare 1 type_pair aligned,
          2 type bit(18) unaligned,
          2 trap_relp bit(18) unaligned,
          2 segname_relp bit(18) unaligned,
          2 offsetname_relp bit(18) unaligned;
```

1) type — assumes a value from 1 to 6:

1 — this is a self-referencing link (i.e., the segment in which the external symbol is located is the very object segment containing this definition) of the form

myself|0+expression, modifier

2 — unused; it was earlier used to define a now obsolete ITB-type link.

3 — this is a link referencing a specified reference name but no symbolic offset name, of the form

refname|0+expression,modifier

4 this is a link referencing both a symbolic reference name and a symbolic offset name, of the form

refname$offsetname+expression,modifier

5 this a self-referencing link having a symbolic offset name, of the form

myself$offsetname+expression,modifier

6 same as type 4 except that the external item will be created if it not found. (See the MPM Subsystem Writers' Guide section, Dynamic Linking.)

2) trap_relp if nonzero then this is a pointer (relative to the base of the definition section) to either a trap pair (if type ¬=6) or to an initialization structure (if type = 6).

3) segname_relp is a code or a pointer depending on the value of type. For types 1 and 5, this item is a code which may assume one of the following values, designating the sections of the self-referencing object segment:

0 self-reference to the object's text section; such a reference is represented symbolically as "*text".

1 self-reference to the object's linkage section; such a reference is represented symbolically as "*link".

2 self-reference to the object's symbol section; such reference is represented symbolically as "*symbol".

For types 3, 4 and 6, this item is a pointer (relative to the base of the definition section) to an aligned acc string containing the reference name portion of an external reference. (See the MPM Reference Guide section, Constructing and Interpreting Names).

4) offsetname_relp has a meaning depending on the value of type. For types 1 and 3, this value is ignored and must be zero. For types 4, 5 and 6, this item is a pointer (relative to the base of the definition section) to an aligned acc string of an external reference. (See the MPM Reference Guide section, Constructing and Interpreting Names, for a discussion of offset names).

The Trap Pair

The trap pair is a structure which specifies a trap procedure to be called before the link associated with the trap pair is resolved by the dynamic linking mechanism. It consists of relative pointers to two links. (Links are defined in the MPM Subsystem Writers' Guide section, The Structure of the Linkage Section.) The first link defines the entry point in the trap procedure to be called. The second link defines a block of information of interest to the trap procedure. It will be passed as one of the arguments of the trap procedure. For more detailed information on trap procedures see the MPM Subsystem Writers' Guide section, Dynamic Linking. The trap pair is structured as follows:

```
declare 1 trap_pair aligned,
          2 entry_relp bit(18) unaligned,
          2 info_relp bit(18) unaligned;
```

1) entry_relp is a pointer (relative to the base of the linkage section) to a link defining the entry point of the trap procedure.

2) info_relp is a pointer (relative to the base of the linkage section) to a link defining information of interest to the trap procedure.

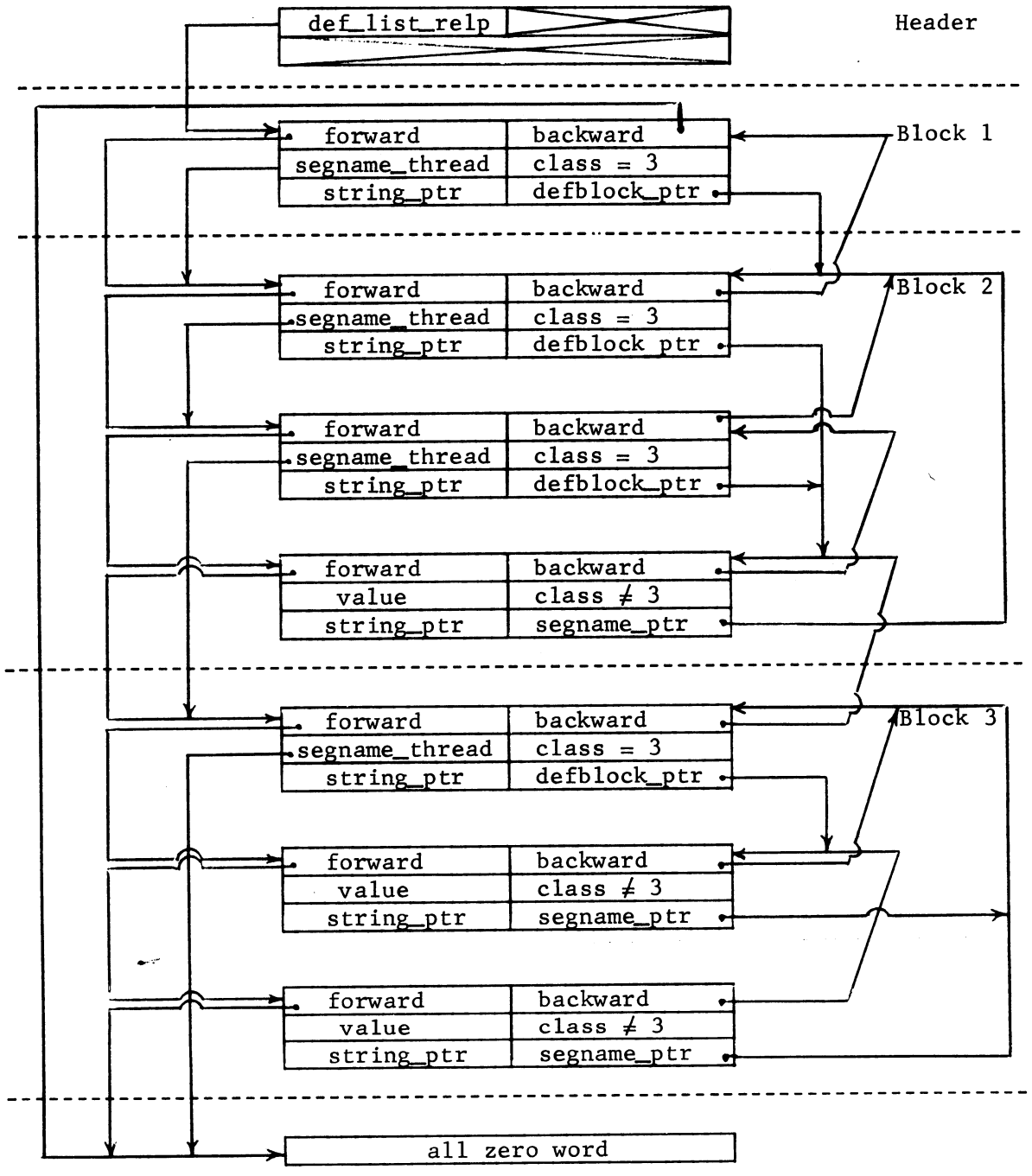


Figure 1: Sample Definition List

THE STRUCTURE OF THE LINKAGE SECTION

The linkage section is substructured into four distinct components, which are a) a fixed-length header which always resides at the base of the linkage section, b) a variable length area used for internal storage, c) a variable length structure of links, and d) an array of first-reference traps. These four components are located within the linkage section in the following sequence:

header
internal storage
links
traps

The length of the linkage section must be an even number of words; in addition, the link structure must begin at an even location (offset) within the linkage section.

The Linkage Section Header

The header of the linkage section has the following format:

```
declare 1 linkage_header aligned,
          2 pad bit(36),
          2 def_section_relp bit(18) unaligned,
          2 first_reference_relp bit(18) unaligned,
          2 obsolete_ptr ptr,
          2 original_linkage_ptr ptr,
          2 links_relp bit(18) unaligned,
          2 linkage_section_length bit(18) unaligned,
          2 object_seg bit(18) unaligned,
          2 obsolete_length bit(18) unaligned;
```

Important: When an object segment is first referenced in a process its linkage section is copied into a per-process data base. At this time certain items in the header are initialized. Items not explicitly described as being initialized by the linker are set by the program which generates the object segment. In addition, the first two words of the header (containing the items pad, def_section_relp, and first_reference_relp) are overwritten with a pointer to the beginning of the object segment's definition section. For more information see the MPM Subsystem Writer's Guide sections, Dynamic Linking, and Standard Stack and Linkage Area Formats.

1) pad is reserved for future use and must be zero.

2) def_section_relp	is a pointer (relative to the base of the object segment) to the base of the definition section.
3) first_reference_relp	is a pointer (relative to the base of the linkage section) to the array of first-reference traps. As explained below, these traps are activated by the linker when the first reference to this object segment is made within a given process. If the value of this item is "0"b then there are no first-reference traps.
4) obsolete_ptr	for historical reasons, linkage sections are sometimes threaded together to form a linkage list. This variable is a pointer to the next linkage section on the thread. This variable is described for completeness; it is not intended for general use.
5) original_linkage_ptr	is a pointer to the original linkage section within the object segment. It is used by the link unsnapping mechanism. It is initialized by the linker.
6) links_relp	is a pointer (relative to the base of the linkage section) to the first link (the base of the link structure).
7) linkage_section_length	is the length in words of the linkage section.
8) object_seg	is the segment number of the object segment. It is initialized by the linker.
9) obsolete_length	when several linkage sections are combined into a list, this item (in the first linkage section in the list) contains the length of the entire list. See the above discussion under obsolete_ptr.

The Internal Storage Area

The internal storage area is an array of words used by compilers to allocate internal static variables and has no predetermined structure.

The Links

This is an array of links, each defining an external name referenced by this object segment whose effective address is unknown at compile time and can be resolved only at the moment of execution. Figure 1 illustrates the structure of a link.

A link must reside on an even location in memory, and must therefore be located at an even offset from the base of the linkage section. The format of a link is:

```
declare 1 link aligned,
          2 header_relp bit(18) unaligned,
          2 ignore1 bit(12) unaligned,
          2 tag bit(6) unaligned,
          2 expression_relp bit(18) unaligned,
          2 ignore2 bit(12) unaligned,
          2 modifier bit(6) unaligned;
```

1) header_relp — is a pointer (relative to the link itself) to the head of the linkage section. It is, in other words, the negative value of the link pair's offset within the linkage section.

2) ignore1 — is reserved for future use and must be "0"b.

3) tag — is a constant (46)8 which represents a hardware fault tag 2 and distinctly identifies an unsnapped link. The snapped link (ITS pair) has a distinct (43)8 tag. See the MPM Subsystem Writers' Guide section, Hardware and Simulated Fault Assignments.

4) expression_relp — is a pointer (relative to the base of the definition section) to the expression structure defining this link.

5) ignore2 — is reserved for future use and must be "0"b.

6) modifier — is a hardware address modifier.

The First-Reference Traps

It is sometimes necessary to perform certain types of initialization of an object segment when it is first referenced for execution (i.e., linked to) in a given process -- for example, to store some per-process information in the segment before it is used. The first-reference trap mechanism provides this facility. For example, the status code assignment mechanism uses this facility. See the MPM Reference Guide section, Handling of Unusual Occurrences.

A first-reference trap consists of two relative pointers. The first points to a link defining the first reference procedure entry point to be invoked. The second points to a link defining a block of information to be passed as an argument to the first-reference procedure. Normally an object segment will have at most one first-reference trap. Bound segments may have up to one per component object segment. For more details on first-reference traps, see the MPM Subsystem Writer's Guide section, Dynamic Linking.

```
declare 1 fr_traps aligned,
          2 decl_vers fixed bin initial(1),
          2 n_traps fixed bin,
          2 array(n_traps) aligned,
            3 call_relp bit(18) unaligned,
            3 info_relp bit(18) unaligned;
```

1) decl_vers — is the version number of the structure.

2) n_traps — specifies the number of trap pointers in this structure.

3) call_relp — is a pointer (relative to the base of the linkage section) to a link defining a procedure to be invoked by the linker upon first reference to this object within a given process.

4) info_relp — if not equal to "0"b, this is a pointer (relative to the base of the linkage section) to a link specifying a block of information to be passed as an argument to the first reference procedure.

Initialization Structure for Type 6 Links

This structure specifies how a link target should be initialized if it is grown because of a type 6 link. It has the following format:

```
declare 1 initialization_info aligned,
          2 n_words fixed bin,
          2 code fixed bin,
          2 info (n_words) bit(36) aligned;
```

1) n_words — is the number of words to increase the size of the segment for the new variable.

2) code — indicates what type of initialization is to be performed. It can have one of the following values:

   0 — no initialization is to be performed.

   3 — copy the info array into the newly grown variable.

   4 — initialize the variable as an area.

3) info — is the image to be copied into the new variable. It exists only if code is 3.

Structure Of The Linkage Section
Standard Object Segment
Page 6

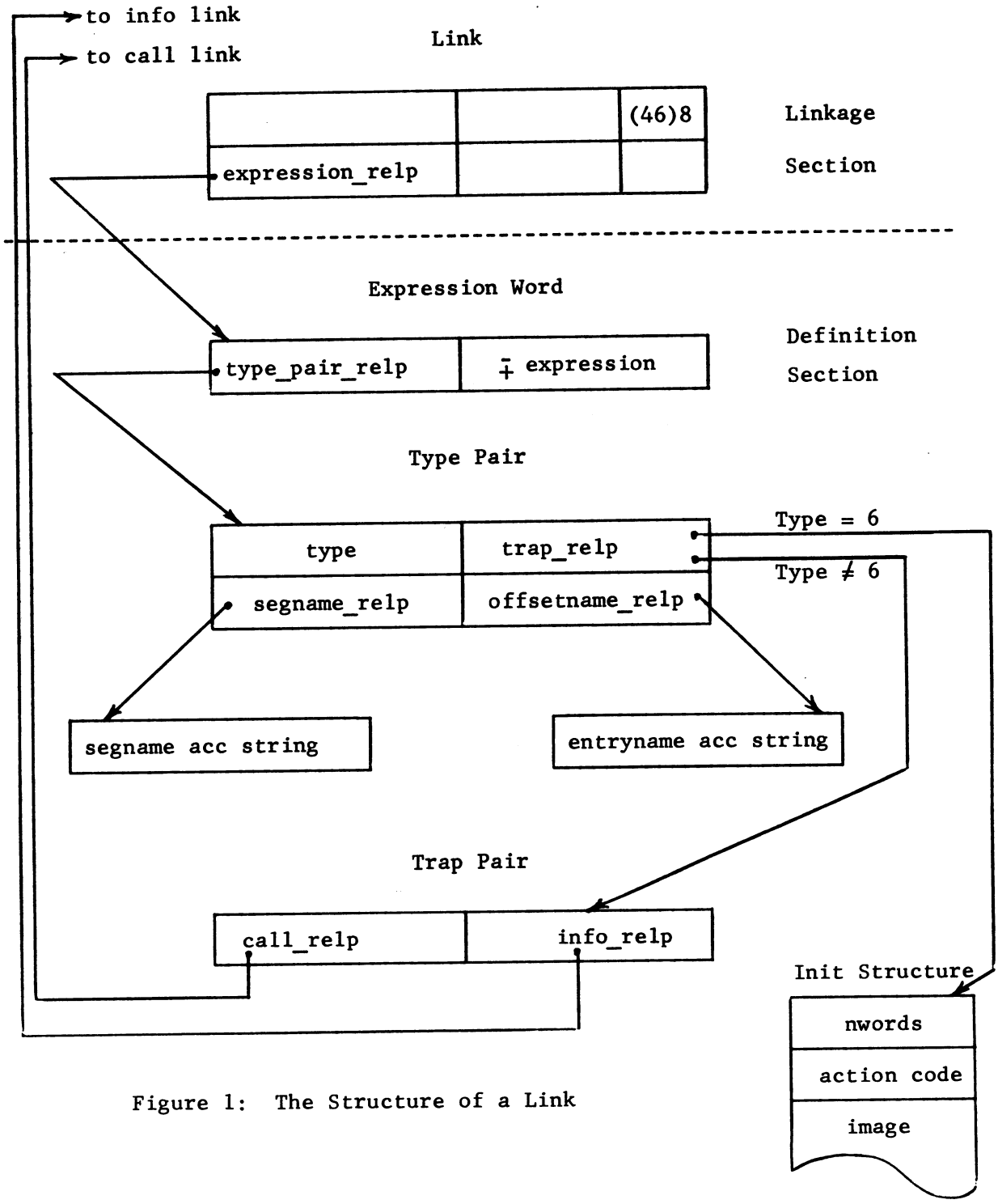


Figure 1: The Structure of a Link

THE STRUCTURE OF THE SYMBOL SECTION

The symbol section consists of one or more symbol blocks threaded together to form a single list. A symbol block has two main functions: 1) to document the circumstances under which the object was created, and 2) to serve as a repository for information which does not belong in any of the other three sections (relocation information, compiler's symbol tree, etc.). The symbol section must contain at least one symbol block, describing the creation circumstances of the object segment. A symbol section may contain more than a single symbol block, for example, in the case of a bound segment where in addition to the symbol block describing the segment's creation by the binder, there is also a symbol block for each of the component object segments.

A symbol block consists of a fixed length header and a variable length area pointed to by the header. The contents of this area depend on the symbol block. For example, a compiler's symbol block may contain a symbol tree, and the binder's symbol block contains the bind map. See the appropriate MPM Subsystem Writer's Guide section for the contents of a symbol block for a PL/I, FORTRAN or ALM object segment.

The Symbol Block Header

All symbol blocks have a standard fixed format header. Not all items in the header have meaning for all symbol blocks. The description of a particular symbol block lists which items have meaning for that symbol block. The header has the following format:

```
declare 1 symbol_block_header aligned,
          2 decl_vers fixed bin initial(1),
          2 identifier char(8) aligned,
          2 gen_version_number fixed bin,
          2 gen_creation_time fixed bin(71),
          2 object_creation_time fixed bin(71),
          2 generator char(8) aligned,
          2 gen_version_name_relp bit(18) unaligned,
          2 gen_version_name_length bit(18) unaligned,
          2 userid_relp bit(18) unaligned,
          2 userid_length bit(18) unaligned,
          2 comment_relp bit(18) unaligned,
          2 comment_length bit(18) unaligned,
          2 text_boundary bit(18) unaligned,
          2 stat_boundary bit(18) unaligned,
          2 source_map_relp bit(18) unaligned,
          2 area_relp bit(18) unaligned,
```

```
        2 section_relp bit(18) unaligned,
        2 block_size bit(18) unaligned,
        2 next_block_thread bit(18) unaligned,
        2 text_relocation_relp bit(18) unaligned,
        2 def_relocation_relp bit(18) unaligned,
        2 link_relocation_relp bit(18) unaligned,
        2 symbol_relocation_relp bit(18) unaligned,
        2 default_truncate bit(18) unaligned,
        2 optional_truncate bit(18) unaligned;
```

1) decl_vers — is the version number of the structure.

2) identifier — is a symbolic name identifying the type of symbol block.

3) gen_version_number — is a code designating the version of the generator which created this object segment. A generator's version number is normally changed when the generator or its output is significantly modified.

4) gen_creation_time — is a calendar clock reading specifying the date/time at which this generator was created.

5) object_creation_time — is a calendar clock reading specifying the date/time at which this symbol block was generated.

6) generator — is the name of the processor which generated this symbol block.

7) gen_version_relp — is a pointer (relative to the base of the symbol block) to an aligned string describing the generator's version. For example,

> "PL/1 Compiler Version 7.3
> of Wednesday, July 28, 1971"

The integer part of the version number imbedded in the string must be identical with the number stored in gen_version_number; the optional fraction as displayed above (7.3) is added in the increments of (.1)

whenever (for reasons such as fixed bugs or minor improvements) a generator is installed which does not differ in any significant way from other generators of the version. It is mandatory that the generator name be updated whenever a new generator is installed for public use.

8) gen_version_name_length is the length of the above string.

9) userid_relp is a pointer (relative to the base of the symbol block) to an aligned string containing the access ID (i.e., the value returned by the subroutine get_group_id_) of the user on whose behalf this symbol block was created.

10) userid_length is the length of the above string.

11) comment_relp is a pointer (relative to the base of the symbol block) to an aligned string containing generator dependent symbolic information. For example, a compiler might store diagnostic messages concerning non-fatal errors encountered while generating the object segment. A value of "0"b indicates no comment.

12) comment_length is the length of the above string.

13) text_boundary is a number indicating what boundary the text section must begin on. For example, a value of 32 would indicate that the text section must begin on a 0 mod 32 word boundary. This value must be a multiple of 2. It is used by the binder to determine where to locate the text section of this object segment.

14) stat_boundary is the same as text_boundary except that it applies to the internal static area of the linkage section of this object segment.

15) source_map_relp — is a pointer (relative to the base of the symbol block) to the source map. (see The Source Map below).

16) area_relp — is a pointer (relative to the base of the symbol block) to the variable length area of the symbol block. The contents of this area depend on the symbol block.

17) section_relp — is a pointer (relative to base of the symbol block) to the base of the symbol section; that is, the negative of the offset of the symbol block in the symbol section.

18) block_size — is the size of the symbol block (including the header) in words.

19) next_block_thread — is a thread (relative to the base of the symbol section) to the next symbol block. This item is "0"b for the last block.

20) text_relocation_relp — is a pointer (relative to the base of the symbol block) to text section relocation information (see The Relocation Information below).

21) def_relocation_relp — is a pointer (relative to the base of the symbol block) to definition section relocation information.

22) link_relocation_relp — is a pointer (relative to the base of the symbol block) to linkage section relocation information.

23) symbol_relocation_relp — is a pointer (relative to the base of the symbol block) to symbol section relocation information.

24) default_truncate — is an offset (relative to the base of the symbol block) starting from which the binder systematically truncates control information (such as relocation bits) from the symbol section, while still maintaining such information as the symbol tree.

25) optional_truncate — is an offset (relative to this base of the symbol block) starting from which the binder may optionally truncate nonessential parts of the symbol tree in order to achieve maximum reduction in the size of a bound object segment.

The Source Map

The source map is a structure which uniquely identifies the source segments used to generate the object segment. It has the following format:

```
declare 1 source_map aligned,
          2 decl_vers fixed bin initial(1),
          2 size fixed bin,
          2 map (size) aligned,
            3 pathname_relp bit(18) unaligned,
            3 pathname_length bit(18) unaligned,
            3 uid bit(36) aligned,
            3 dtm fixed bin(71);
```

1) decl_vers — is the version number of the structure.

2) size — is the number of entries in the map array; that is, the number of source segments used to generate this object segment.

3) pathname_relp — is a pointer (relative to the base of the symbol block) to an aligned string containing the full absolute pathname of this source segment.

4) pathname_length — is the length of the above string.

5) uid — is the unique identifier of this source segment at the time the object segment was generated.

6) dtm — is the date-time modified of this source segment at the time the object segment was created.

The Relocation Information

The relocation information designates all instances of relative addressing within a given section of the object segment, so as to enable the relocation of such a section (as in the case of binding). A variable length prefix coding scheme is used, where there is a logical relocation item for each halfword of a given section. If the halfword is an absolute value (nonrelocatable), that item is a single bit whose value is zero. Otherwise, the item is a string of either 5 or 15 bits whose first bit is set to "1"b. The relocation information is concatenated to form a single string which may only be accessed sequentially; if the next bit is a zero, it is a single-bit absolute relocation item, otherwise it is either a 5- or a 15-bit item depending upon the relocation codes as defined below.

There are four distinct blocks of relocation information, one for each of the four object segment sections: text, definition, linkage and symbol; these relocation blocks are known as rel_text, rel_def, rel_link and rel_symbol, correspondingly.

The relocation blocks reside within the symbol block of the generator which produced the object segment. The correspondence between the packed relocation items and the halfwords in a given section is made by matching the sequence of items with a sequence of halfwords, from left to right and from word to word by increasing value of address.

The relocation block pointed to from the symbol block header (e.g., rel_text) is structured as follows.

```
declare 1 relinfo aligned,
          2 decl_vers fixed bin initial(2),
          2 n_bits fixed bin,
          2 relbits bit(n_bits) aligned;
```

1) decl_vers — is the version number of the structure.

2) n_bits — is the length (in bits) of the string of relocation bits.

3) relbits — is the string of relocation bits.

Following is a tabulation of the possible codes and their corresponding relocation types, followed by a description of each relocation type.

"0"b	-	Absolute
"10000"b	-	Text
"10001"b	-	Negative Text
"10010"b	-	Link 18
"10011"b	-	Negative Link 18
"10100"b	-	Link 15
"10101"b	-	Definition
"10110"b	-	Symbol
"10111"b	-	Negative Symbol
"11000"b	-	Internal Storage 18
"11001"b	-	Internal Storage 15
"11010"b	-	Self Relative
"11011"b	-	Unused
"11100"b	-	Unused
"11101"b	-	Unused
"11110"b	-	Expanded Absolute
"11111"b	-	Escape

1) Absolute — Do not relocate.

2) Text — Use text section relocation counter.

3) Negative Text — Use text section relocation counter. The reason for having distinct relocation codes for negative quantities is that special coding might have to be used in order to convert the 18-bit field in question into its correct fixed binary form.

4) Link 18 — Use linkage section relocation counter on the entire 18-bit halfword. This, as well as the Negative Link 18 and the Link 15 relocation codes apply only to the array of links in the linkage section (i.e., by definition, usage of these relocation codes implies external reference through a link).

5) Negative Link 18 — Same as Link 18 above.

6) Link 15 — Use linkage section relocation counter on the low order 15 bits of the halfword. This relocation code may only be used in conjunction with

	an instruction featuring a base/offset address field.
7) Definition	Indicates that the halfword contains an address which is relative to the base of the definition section.
8) Symbol	Use symbol section relocation counter.
9) Negative Symbol	Same as Symbol above.
10) Internal Storage 18	Use internal storage relocation counter on the entire 18-bit halfword.
11) Internal Storage 15	Use internal storage relocation counter on the low order 15 bits of the halfword.
12) Self Relative	Indicates that the halfword contains a relocatable address which is referenced using a location counter modifier; the instruction is selfrelocating.
13) Expanded Absolute	It has been established that a major part of an object program has the absolute relocation code; for efficiency reasons, the expanded absolute code allows the definition of a block of absolutely relocated halfwords. The five bits of relocation code are immediately followed by a fixed length 10-bit field which is a count of the number of contiguous halfwords all having an absolute relocation. Obviously, usage of the expanded absolute code can be economically justified only if the number of contiguous absolute halfwords exceeds 15.
14) Escape	Reserved for possible future use.

 (END)

THE STRUCTURE OF THE OBJECT MAP

The object map contains information which allows the various sections of an object segment to be located. The map itself may be located immediately before or immediately after any one of the five sections. Translators normally place it immediately after the symbol section. The last word of the segment must contain a left-justified 18-bit pointer (relative to the base of the object segment) to the object map. The object map has the following format:

```
declare 1 object_map aligned,
          2 decl_vers fixed bin init(1),
          2 identifier char(8) aligned,
          2 text_relp bit(18) unaligned,
          2 text_length bit(18) unaligned,
          2 def_relp bit(18) unaligned,
          2 def_length bit(18) unaligned,
          2 link_relp bit(18) unaligned,
          2 link_length bit(18) unaligned,
          2 symb_relp bit(18) unaligned,
          2 symb_length bit(18) unaligned,
          2 bmap_relp bit(18) unaligned,
          2 bmap_length bit(18) unaligned,
          2 format aligned,
            3 bound bit(1) unaligned,
            3 relocatable bit(1) unaligned,
            3 procedure bit(1) unaligned,
            3 standard bit(1) unaligned,
            3 unused bit(14) unaligned;
```

1) decl_vers — is the version number of the structure.

2) identifier — is the constant "obj_map".

3) text_relp — is a pointer (relative to the base of the object segment) to the base of the text section.

4) text_length — is the length (in words) of the text section.

5) def_relp — is a pointer (relative to the base of the object segment) to the base of the definition section.

6) def_length — is the length (in words) of the definition section.

7) link_relp	is a pointer (relative to the base of the object segment) to the base of the linkage section.
8) link_length	is the length (in words) of the linkage section.
9) symb_relp	is a pointer (relative to the base of the object segment) to the base of the symbol section.
10) symb_length	is the length (in words) of the symbol section.
11) bmap_relp	is a pointer (relative to the base of the object segment) to the base of the break map section.
12) bmap_length	is the length (in words) of the break map section.
13) bound	is "1"b if the object segment is a bound segment.
14) relocatable	is "1"b if the object segment is relocatable; that is, if it contains relocation information. This information (if present) must be stored in the segments' first symbol block. See the MPM Subsystem Writers' Guide section, The Structure of the Symbol Section.
15) procedure	is "1"b if this is an executable object segment.
16) standard	is "1"b if the object segment is in standard format.
17) unused	is reserved for future use and must be "0"b.

 (END)

CONVENTIONS ON GENERATED CODE

This document describes those parts of the generated code which must conform to a systemwide standard. For a description of the various relocation codes see the MPM Subsystem Writers' Guide section, The Structure of the Symbol Section.

The Text Section

Those parts of the text section which must conform to a systemwide standard are the entry sequence and text relocation codes.

The Entry Sequence

The entry sequence must fulfil two requirements: 1) that at the location preceding the entrypoint (i.e., entrypoint-1) there is a left adjusted 18-bit relative pointer to the definition of that entrypoint (within the definition section); and 2) that the entry sequence executed within that entrypoint store an ITS pointer to that entrypoint at sp|22 so that by inspecting the procedure's current stack frame one may determine the address of the entrypoint at which it was invoked, and then reconstruct that entry's symbolic name through use of its definition pointer.

Text Relocation Codes

The following list defines the only relocation codes which may be generated in conjunction with the text section, and then only within the scope of the restrictions specified.

Absolute	no restriction.
Text	no restriction.
Negative Text	no restriction.
Link 18	may only be direct (i.e., unindexed) reference to a link.
Link 15	may only appear within the address field of a (base/offset) type instruction (bit 29="1"b). The instruction must not contain a "10"b tm modifier, and may be indexed (i.e., specify an index register) only if it has a "11"b tm modifier. Also, the following instruction codes may not have this relocation code:

STBA (551)8
STBQ (552)8
STCA (751)8
STCQ (752)8

Definition	no restriction.
Symbol	no restriction.
Internal Storage 18	no restriction.
Internal Storage 15	may only apply to the left half of a word. If the word is an instruction, it must not contain a "10"b tm modifier.
Self Relative	no restriction.
Expanded Absolute	no restriction.

The peculiar restrictions imposed upon the Link 15 and Internal Storage 15 relocation codes stem from the fact that these relocation codes apply to base/offset type address fields encountered in the address portion of machine instructions; the effective value of such an address is computed by the hardware at execute time. To that end, certain hardware restrictions are imposed on such instructions. When the Multics binder processes these instructions, it often resolves them into simple-address format and has to further modify information in the opcode (right hand) portion of the instruction word. Therefore, these relocation codes must only be specified in a context which is comprehensible to the Honeywell 6180 control unit.

The Definition Section

Definition relocation codes and implicit definitions are the parts of the definition section which must conform to a systemwide standard.

Definition Relocation Codes

Absolute	no restriction.
Text	no restriction.
Link 18	no restriction.
Definition	no restriction.

Symbol	no restriction.
Internal Storage 18	no restriction.
Self Relative	no restriction.
Expanded Absolute	no restriction.

Implicit Definitions

All generated object segments must feature the following implicit definition,

"symbol_table" defining the base of the symbol block generated by the current language processor, relative to the base of the symbol section.

The Linkage Section

Those parts of the linkage section which must conform to a systemwide standard are described below.

The Internal Storage

The internal storage is a repository for items of the internal static storage class. It may contain data items only; it may not contain any executable code.

The Links

The link area may only contain a set of links. The links must be considered as distinct unrelated items, and no structure (e.g., array) of links may be assumed. They must be accessed explicitly and individually through an unindexed internal reference featuring the Link 18 or the Link 15 relocation codes.

Linkage Relocation Codes

Only the linkage section header and the links may have relocation codes associated with them (the internal storage area has associated with it a single Expanded Absolute relocation item).

Absolute	no restriction; mandatory for the internal storage area.

Text	no restriction.
Link 18	no restriction.
Negative Link 18	no restriction.
Definition	no restriction.
Internal Storage 18	no restriction.
Expanded Absolute	no restriction.

The Symbol Section

The symbol section may contain information related to some other section (such as a symbol tree defining addresses of symbolic items), and therefore may have relocation codes associated with it. They are:

Absolute	no restriction.
Text	no restriction.
Link 18	no restriction.
Definition	no restriction.
Symbol	no restriction.
Negative Symbol	no restriction.
Internal Storage 18	no restriction.
Self Relative	no restriction.
Expanded Absolute	no restriction.

STRUCTURE OF BOUND SEGMENTS

A bound segment is derived from several object segments which have been combined in order to have all the internal intersegment references automatically prelinked and to reduce the combined size by minimizing page breakage. The component segments are not merely concatenated, however; the binder breaks them apart and creates an object segment with single text, definition, linkage and symbol sections as illustrated in Figure 1. As will be explained below, the definition section and link array are completely reconstructed while the text, internal static and symbol sections are simply the corresponding concatenations of the component segments' text, internal static and symbol sections, with relocation adjustments. (See the MPM Subsystem Writers' Guide section, the Structure of the Symbol Section.) In addition, a symbol block for the binder is included in the symbol section.

Prelinking

The most important differences between bound and unbound groups of segments are the ways they reference external items and the ways they can be referenced. Most references by one component to another component in the same bound segment are prelinked; i.e. the link references are converted to direct text-to-text transfers and the associated links are not regenerated. The remaining external links are coalesced so that for the whole bound segment there is only one link for each different target. Prelinking enables some component segments to lose their identity in cases where the bound segment itself is the main logical entity, having been coded as separate segments for ease of coding and debugging. Definitions for external entries which are no longer necessary, i.e., which have became completely internal, may be omitted from the bound segment (see the MPM write-up for the bind command).

Definition Section

The definition section of a bound segment is generally more elaborate than that of an unbound object segment because it reflects both the combination and deletion of definitions. There is a definition block for each component that has any externally referenceable items left. It contains the retained definitions and the segment names associated with the component. This organization allows definitions for multiple entries with the same name to be distinguished. The first definition block is for the binder and contains a definition for bind_map, which is discussed below.

Structure of Bound Segments
Standard Object Segment
Page 2

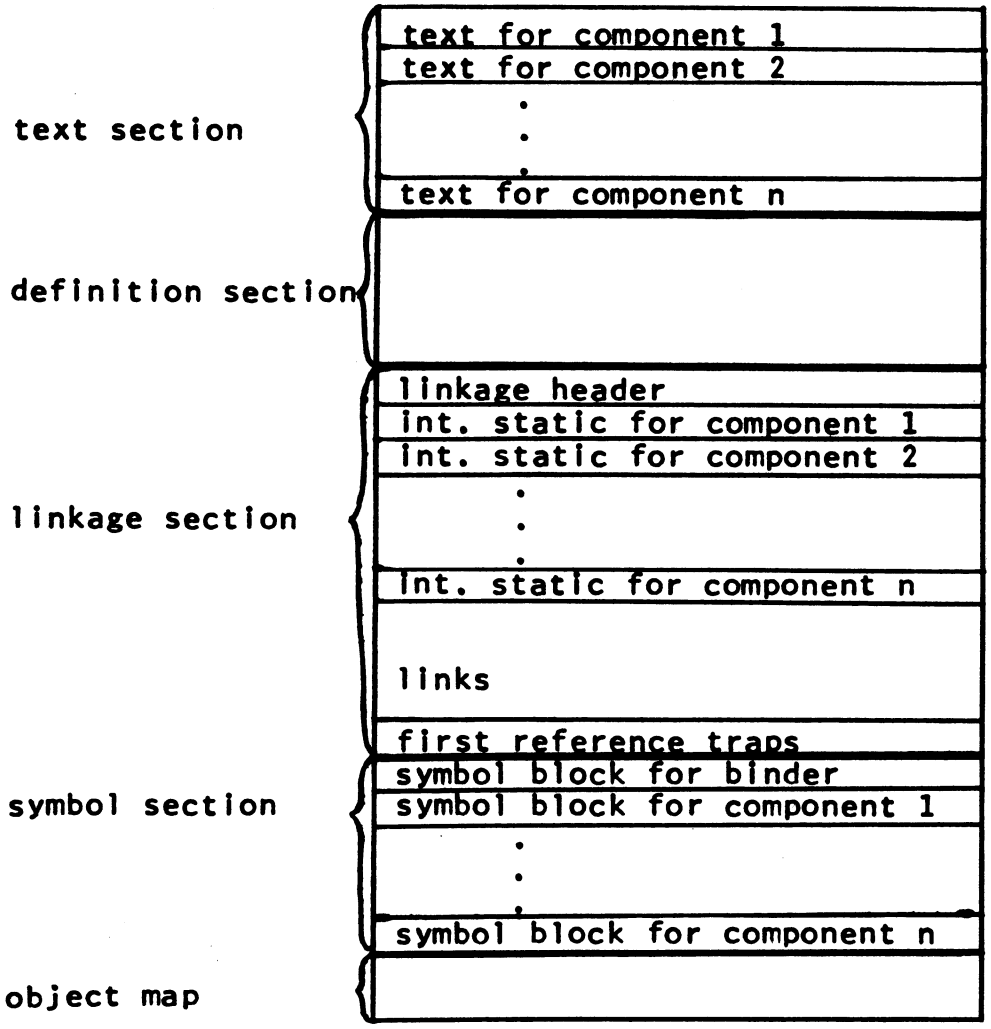


Figure 1: The Structure of a Bound Segment.

The Binder's Symbol Block

The binder's symbol block has a standard header if all of the components are standard object segments. The symbol block can be located via the bind_map definition. Most of the items in the header are adequately explained in the MPM Subsystem Writers' Guide section, The Structure of the Symbol Section; however, some have special meaning for bound segments. The format of a standard symbol block header is repeated below for reference, followed by the explanations specific to the binder's symbol block.

```
declare 1 symbol_block_header aligned,
          2 decl_vers fixed bin initial(1),
          2 identifier char(8) aligned,
          2 gen_version_number fixed bin,
          2 gen_creation_time fixed bin(71),
          2 object_creation_time fixed bin(71),
          2 generator char(8) aligned,
          2 gen_version_name_relp bit(18) unaligned,
          2 gen_version_name_length bit(18) unaligned,
          2 userid_relp bit(18) unaligned,
          2 userid_length bit(18) unaligned,
          2 comment_relp bit(18) unaligned,
          2 comment_length bit(18) unaligned,
          2 text_boundary bit(18) unaligned,
          2 stat_boundary bit(18) unaligned,
          2 source_map_relp bit(18) unaligned,
          2 area_relp bit(18) unaligned,
          2 section_relp bit(18) unaligned,
          2 block_size bit(18) unaligned,
          2 next_block_thread bit(18) unaligned,
          2 text_relocation_relp bit(18) unaligned,
          2 def_relocation_relp bit(18) unaligned,
          2 link_relocation_relp bit(18) unaligned,
          2 symbol_relocation_relp bit(18) unaligned,
          2 default_truncate bit(18) unaligned,
          2 optional_truncate bit(18) unaligned;
```

2) identifier — is the string "bind_map".

6) generator — is the string "binder".

11) comment_relp — is currently always "0"b.

16) area_relp — is a pointer (relative to the base of the symbol block) to the beginning of the bind map. (See below for a description of the bind map.)

Bound segments currently are not relocatable, so none of the relocation relative pointers or truncation offsets have any meaning.

The Bind Map

The bind map is part of the symbol block produced by the binder and describes the relocation values assigned to the various sections of the bound component object segments. It consists of a variable length structure followed by an area in which variable length symbolic information is stored. The bind map structure has the following format:

```
declare 1 bindmap based aligned,
          2 decl_vers fixed bin initial(1),
          2 n_components fixed bin,
          2 component(n_components) aligned,
            3 name_relp bit(18) unaligned,
            3 name_length bit(18) unaligned,
            3 generator_name char(8) aligned,
            3 text_relp bit(18) unaligned,
            3 text_length bit(18) unaligned,
            3 static_relp bit(18) unaligned,
            3 static_length bit(18) unaligned,
            3 symbol_relp bit(18) unaligned,
            3 symbol_length bit(18) unaligned,
            3 defblock_relp bit(18) unaligned,
            3 number_of_blocks bit(18) unaligned;
```

1) decl_vers — is a constant designating the format of this structure; whenever the structure is modified, so is this constant, allowing system tools to easily differentiate between several incompatiable versions of a single structure.

2) n_components — is the number of component object segments bound within this bound segment.

3) component — is a variable-length array featuring one entry per bound component object segment.

4) name_relp	is a pointer (relative to the base of the bindmap structure) to the symbolic name of the bound component. This is the name under which the component object was identified within the archive file used as the binder's input (i.e., the name corresponding to the object's objectname entry in the bindfile).
5) name_length	is the length (in characters) of the component's name.
6) generator_name	is the name of the translator which created this component object segment.
7) text_relp	is a pointer (relative to the base of the bound segment) to the component's text section.
8) text_length	is the length (in words) of the component's text section.
9) static_relp	is a pointer (relative to the base of the linkage section) to the component's internal static.
10) static_length	is the length of the component's internal static.
11) symbol_relp	is a pointer (relative to the base of the symbol section) to the component's symbol section.
12) symbol_length	is the length of the component's symbol section.
13) defblock_relp	if non-zero, this is a pointer (relative to the base of the definition section) to the component's definition block (first class 3 segname definition of that component's definition block).
14) number_of_blocks	is the number of symbol blocks in the component's symbol section.

STANDARD STACK AND LINKAGE AREA FORMATS

Because of the complexity of the Multics hardware, the linkage mechanism and the stack manipulations, a series of Multics execution environment standards has been adopted. All standard (supported) translators (including assemblers) must abide by these standards. All supervisor, standard service and development procedures will follow these standards, and further, they will assume that other procedures do so as well.

The Multics Stack

The normal mode of execution in a standard Multics process makes use of a stack segment. There is at least one stack segment for each ring. The stack for a given ring has the entry name stack_R, where R is the ring number, and is located in the process directory. Each stack contains a header followed by as many stack frames as are required by the executing procedures. A stack header consists of pointers to special code and data which are initialized when the stack is created. Some of the pointers in the stack header are variable and change as the process executes. These pointers are in the stack header to insure that the pointers can always be retrieved without supervisor intervention (for efficiency). The actual format of the stack header is described below.

The stack frames (which begin at a location specified in the stack header) are variable in length and contain both control information and data for dynamically active procedures. In general, a stack frame is allocated by the procedure to which it belongs when that procedure is invoked. The stack frames are threaded to each other with forward and backward pointers making it an easy task to trace the stack, for whatever reason, in either direction. The stack discipline described below is critical to normal Multics operation and any deviations from the stated discipline may result in unexpected behavior.

The Stack Header

The stack header contains pointers to (per-ring) information about the process. There also exist pointers to operator segments and code sequences which can be used to invoke the standard call, push, pop, and return functions. (See below.) Figure 1 gives the format of the stack header. The various pointers and variables mentioned there are described below. A PL/I declaration of the stack header follows.

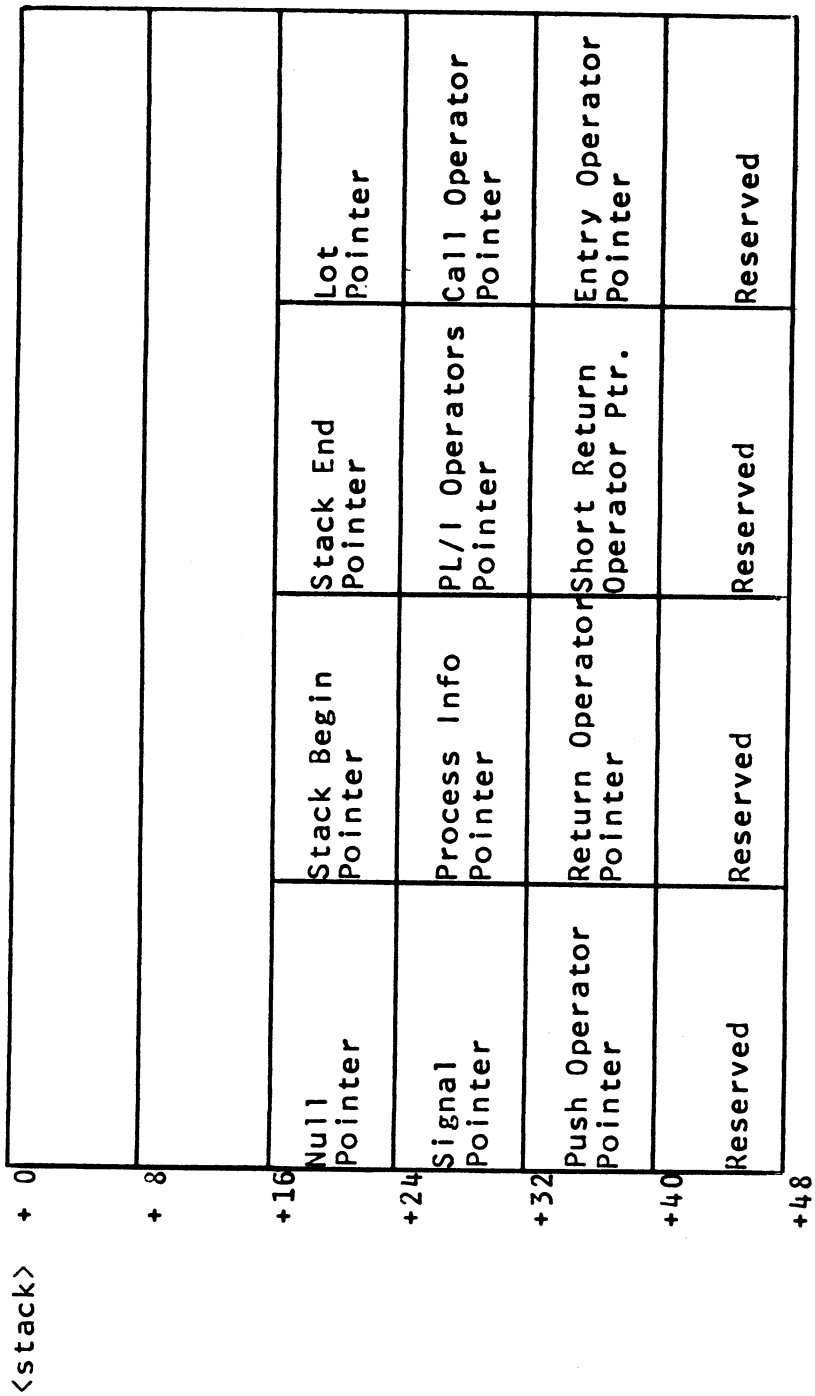


Figure 1: Stack Header Format

```
declare 1 stack_header based (sb) aligned,
          2 pad1 (16) fixed bin,
          2 null_ptr ptr,
          2 stack_begin_ptr ptr,
          2 stack_end_ptr ptr,
          2 lot_ptr ptr,
          2 signal_ptr ptr,
          2 process_info_ptr ptr,
          2 pl1_operators_ptr ptr,
          2 call_op_ptr ptr,
          2 push_op_ptr ptr,
          2 return_op_ptr ptr,
          2 short_return_op_ptr ptr,
          2 entry_op_ptr ptr,
          2 pad2 (8) fixed bin;
```

1) Null Pointer. This field contains a null pointer value. In some circumstances the stack header can be treated as a stack frame. When this is done this null pointer field would occupy the same location as the previous stack frame pointer of the stack frame. (See below.) The null pointer is used to indicate that there is no stack frame prior to the stack header.

2) Stack Begin Pointer. The stack begin pointer is a pointer to the first stack frame on the stack. The first stack frame does not necessarily begin at the end of the stack header. Other information, for example the linkage offset table, may be located between the stack header and the first stack frame.

3) Stack End Pointer. The stack end pointer is a pointer to the first unused word after the last stack frame. It points to the location where the next stack frame will be placed on this stack (if one is needed). Note that is is required that a stack frame be a multiple of 16 words and hence both of the above pointers point to zero (mod 16) word boundaries.

4) Lot Pointer. This is a pointer to the linkage offset table (lot) for the current ring. The lot is a table containing (packed) pointers to the linkage sections known in the ring in which the lot exists. The linkage offset table and combined linkage segment are described below.

5) Signal Pointer. This is a pointer to the signalling procedure to be invoked when a condition is raised in the current ring.

6) Process Info Pointer. This is a pointer to a procedure which may be invoked to return specific variables of interest about the process.

7) PL/I Operators Pointer. This is a pointer into pl1_operators_. It is used by PL/I and FORTRAN object code to locate the appropriate operator segment. The pointer is in the base of the stack to avoid a linkage fault to pl1_operators_ from every PL/I or FORTRAN procedure segment invoked in the process.

8) Call Operator Pointer. This is a pointer to the Multics standard call operator used by ALM procedures. It is used to invoke another procedure in the standard way.

9) Push Operator Pointer. This is a pointer to the Multics standard push operator which is used by ALM programs when allocating a new stack frame. All push's performed on a Multics stack should use either this operator or an equivalent one, or unexpected behavior may result. (The push operation was formerly called save.)

10) Return Operator Pointer. This is a pointer to the Multics standard return operator used by ALM procedures. It assumes that a push has been performed by the invoking ALM procedure and will pop the stack prior to returning control to the caller of the ALM procedure.

11) Short Return Operator Pointer. This is a pointer to the Multics standard short return operator used by ALM procedures. It is invoked by a procedure which has not performed a push to return control to its caller.

12) Entry Operator Pointer. This is a pointer to the Multics standard entry operator. This entry operator currently does little more than find a pointer to the linkage section.

The previous five operators are invoked by the object code generated by the ALM assembler. Other translators which hope to use the standard call/push/return strategy should also use these operators. For a detailed description of what the operators do and how to invoked them see the MPM Subsystem Writers' Guide section, Subroutine Calling Sequences.

The PL/I and FORTRAN compilers do not use these operators. Instead they use slightly different operators which perform equivalent and compatible functions. All supported translators, however, depend on the effects generated by these operators.

Unspecified areas of the stack header should not be used as they are reserved for future expansion.

The Stack Frame

This section describes the format of a standard Multics stack frame. The format should be strictly adhered to in that several critical procedures of the Multics system depend on it. A bad stack segment or stack frame can easily lead to process termination, looping, and other undesirable effects.

In the discussion which follows the concept of owning a stack frame refers to the procedure that created the stack frame (with a push operation). Some programs (generally ALM programs) never perform a push and hence do not own any stack frame. If such a procedure is executing it can neither call another procedure nor use stack temporaries. All stack information refers to the program which called any such program.

Figure 2 below illustrates the detailed structure of a stack frame. The following descriptions are based on that diagram and on the following PL/I declaration.

```
declare 1 stack_frame based (sp) aligned,
          2 pad1 (16) fixed bin,
          2 prev_stack_frame_ptr ptr,
          2 next_stack_frame_ptr ptr,
          2 return_ptr ptr,
          2 entry_ptr ptr,
          2 operator_link_ptr ptr,
          2 argument_ptr ptr,
          2 reserved (2) fixed bin,
          2 on_unit_rel_ptrs (2) bit(18) unaligned,
          2 operator_return_offset bit (18),
          2 pad2 (8) fixed bin;
```

1) Previous Stack Frame Pointer. The previous stack frame pointer is a pointer to the base of the stack frame of the procedure which called the procedure owning the current stack frame. This pointer may or may not point to a stack frame in the same stack segment.

2) Next Stack Frame Pointer. The next stack frame pointer points to the base of the next stack frame. For the last stack frame on a stack the pointer points to the next available area in the stack where a procedure can lay down a stack frame; i.e., it has the same value as the stack end pointer in the stack

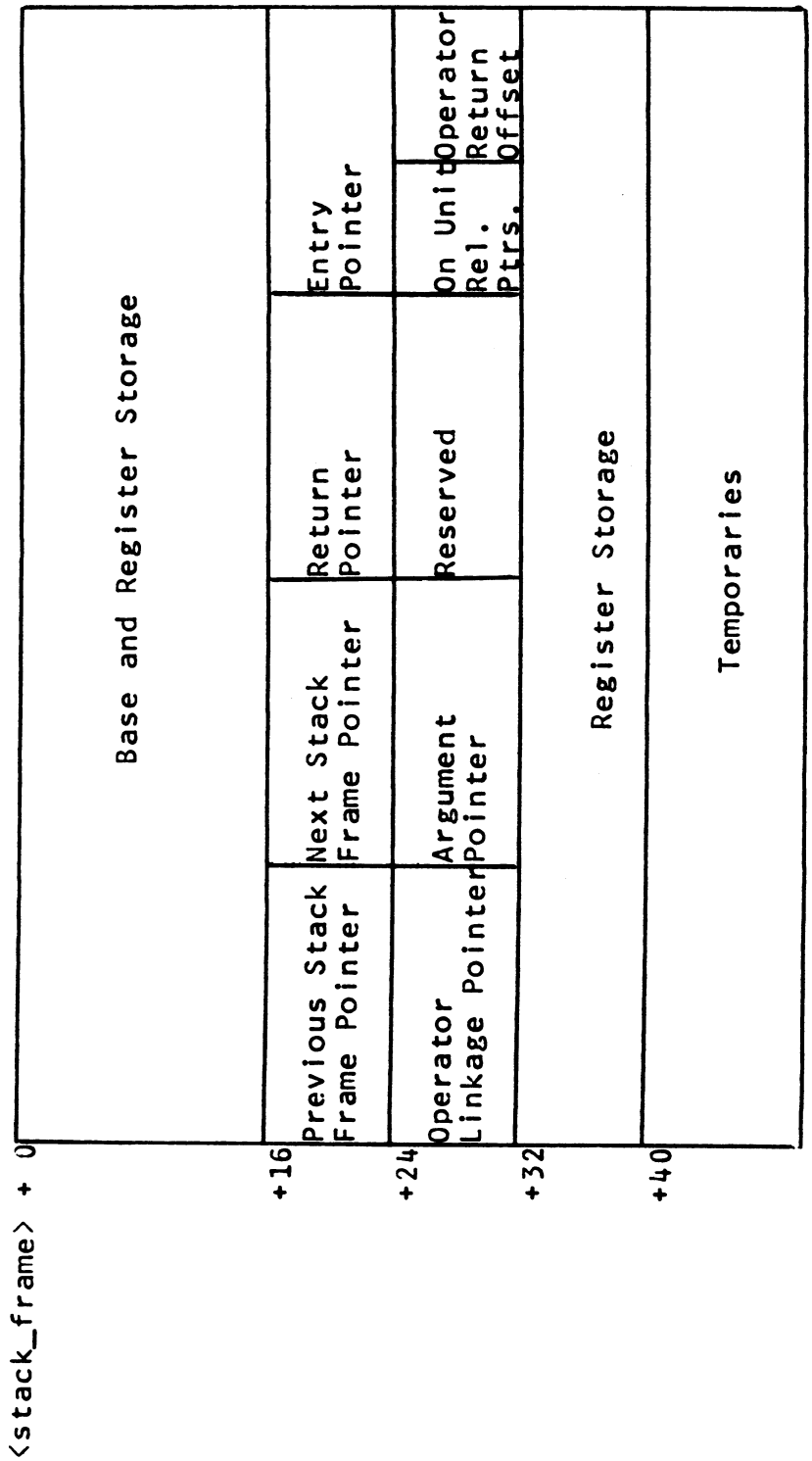


Figure 2: Stack Frame Format

base. The previous stack frame pointers and the next stack frame pointers for the stack frames on a stack form threads through all active frames on the stack. These two threads are used by debugging tools to search and trace the stack.

3) Return Pointer. This is a pointer to the location to be transferred to in order to return to the procedure owning the given frame. This pointer is undefined if the procedure has never made a call out, and points to the return location associated with the last call out if the given procedure has been returned to and is currently executing. (Note: For Version I PL/I programs, this pointer points into a special return location in pl1_operators, the Version I operator segment. See below.)

4) Entry Pointer. This is a pointer to the procedure entry point which was called and which owns the stack frame. The pointer points to a standard entry point. See the MPM Subsystem Writers' Guide section, The Structure of the Text Section.

5) Operator Linkage Pointer. For PL/I and FORTRAN procedures, this is a pointer into the operator segment being used by the procedure owning the given stack frame. For machine language programs, this pointer points to the linkage section for the procedure.

6) Argument Pointer. This is a pointer to the argument list passed to the procedure owning the given stack frame.

7) On Unit Relative Pointers. This word contains two relative pointers to on unit information contained within the stack frame. This on unit information is valid if and only if bit 29 of the second word of the previous stack frame pointer is a 1. (Note that this bit was automatically set to 0 when a push was performed by the procedure owning the stack frame.) The left half of the word is a pointer (relative to the stack frame base) to a list of enabled conditions. The right half of the word is a pointer (relative to the stack frame base) to a list of special entries which are enabled for the current stack frame. These latter entries include default handlers and cleanup procedure information if any were specified by the procedure owning the stack frame.

8) Operator Return Offset. This word contains a return location for certain pl1_operators_ functions. If the left half of the word is nonzero it is a relative pointer to the return location in the compiled program (return from pl1_operators_).

If the word is zero, a dedicated register (known by pl1_operators_) contains the return location.

Further Notes On Stack Frames

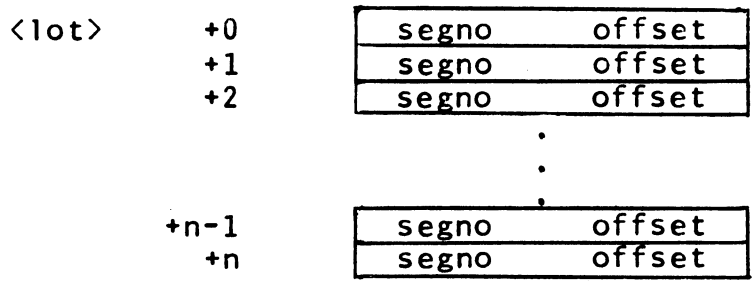
There are two major areas of a stack frame which are not explicitly defined above. The first area consists of the first 16 words of the stack frame and the second area consists of words 32 through 39 of a stack frame. These areas have a well defined purpose for ALM programs and are used internally by the PL/I and FORTRAN programs. The contents of these areas is not, however, always defined or meaningful. The procedure owning the stack frame can use these areas as it sees fit.

For ALM programs the areas are used by the call operator as invoked by the call pseudoop. The pointer registers are stored in the first (16-word) block and the arithmetic registers are stored in the second (8-word) block.

For PL/I and FORTRAN programs it is difficult to clearly state the use of these areas. Version I PL/I programs, for example, store the arithmetic registers at an offset of 8 within the stack frame and use the rest of these areas for internal control information. Version II PL/I programs do not normally store the arithmetic registers anywhere in the stack frame.

Linkage Offset Table

A pointer exists (as described above) in each stack header pointing to the linkage offset table (lot) for the current ring. The lot, in turn, contains packed pointers to linkage sections of segments known in the current ring. A packed pointer to the linkage section for a particular object segment (with segment number N) is found at the N-th entry (numbered from zero) in the lot. Each entry is one word in length with the first 18 bits containing the segment number and the second 18 bits containing the offset of the linkage section. A linkage offset table has the following format:



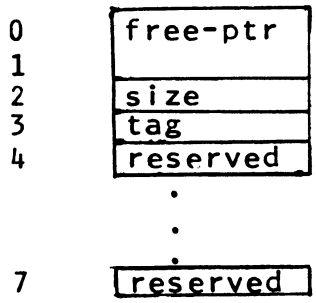
The length of the lot, n, can be specified by the user.

Combined Linkage Area

In general, the linkage sections for all object segments in a given ring of a process are copied into an area known as the combined linkage area. This area may be located in the stack of the given ring and/or a sequence of one or more segments in the process directory with the names combined_linkage_R.nn where R is the given ring and nn is a sequence number ranging from 00 to 99. Such a segment is called a combined linkage segment. Space for the combined linkage area is initially allocated either in the segment combined_linkage_R.00 or in the stack. (The choice of which space is used is under the control of the project administrator.) In either case, if this space is used up a new segment combined_linkage_R.01 is created. In general, when a combined linkage segment is filled up a new one is created with the next higher sequence number.

Each combined linkage segment (and the space in the stack if it is used) consists of a header followed by contiguous blocks, one block for each linkage section. In addition, the linkage offset table may be located in the combined linkage area.

The format of the header is as follows:



It is described by the following pl1 declaration:

```
declare 1 combined_linkage_header aligned,
          2 free_ptr ptr,
          2 size fixed bin,
          2 tag fixed bin,
          2 pad(4) fixed bin;
```

1) free_ptr is a pointer to the first free word in the combined linkage segment or area of stack.

2) size is the maximum size of the combined linkage segment or area of stack.

3) tag is the sequence number of this combined linkage segment.

4) pad is reserved for future use and must not be used by the user.

The linkage offset table entry (see above) of a given object segment usually points to the block in the combined linkage area which contains the linkage section for that segment. Sometimes a separate segment is created for the linkage section of an object segment. In this case the lot entry points to this segment.

For the format of the linkage section see the MPM Subsystem Writers' Guide section, The Structure of the Linkage Section.

Warning

The entry pointer (the fourth item of a stack frame - in words 22 and 23) will not be implemented until Multics is operating on the Honeywell 6180 hardware.

SUBROUTINE CALLING SEQUENCES

This section describes the Multics standard call and return conventions. For information about the format of stack segments and stack frames see the MPM Subsystem Writers' Guide section, Standard Stack and Linkage Area Formats.

There are seven basic functions which must be performed at one time or another in the general case of one procedure calling another and then being returned to. These are:

1) calling a procedure -- i.e., transferring control to the procedure and passing an argument list pointer to the called procedure,

2) generating a linkage pointer for the called procedure,

3) creating a stack frame for the called procedure,

4) saving certain standard items in the stack frame of the called procedure,

5) releasing the stack frame of the called procedure just prior to returning,

6) reestablishing the execution environment of the calling procedure, and

7) returning control to the calling procedure.

Note that preparation of the argument list, although necessary, is not included above because the operators described below need know nothing about the format of an argument list. The argument list format is described below, under The Argument List.

Operators to perform the above functions have been provided in the standard operator segments pl1_operators (for PL/I Version I programs) and pl1_operators_ (for PL/I Version II, FORTRAN, and ALM procedures). These operators are invoked when appropriate by the object code generated by these translators.

This document describes the operators used by ALM procedures. The operators used by PL/I and FORTRAN procedures are basically the same but differ at a detailed level due to: 1) slight changes in the execution environment when PL/I and FORTRAN programs are running; and 2) simplification and combination of operators made possible by PL/I's knowledge of the execution environment. The PL/I and FORTRAN operators will not be

described here other than to define a minimum execution environment which must be established when returning to a PL/I or FORTRAN program.

The following description is, in general, in Honeywell 6180 hardware terms. The same conventions apply on the 645 system with the additional restriction that the normal execution environment assumes the 8 base registers are paired to each other in the normal manner (i.e., even bases are internal, and the pairings are 0-1, 2-3, 4-5, 6-7).

To perform the seven functions mentioned above for ALM procedures, five operators have been provided upon which ALM object code calls.

These operators are:

1) call — performs function 1) above,

2) entry — performs function 2) and part of function 4) above,

3) push — performs function 3) and part (the rest) of function 4) above,

4) return — performs functions 5), 6), and 7) above,

5) short_return — performs functions 6) and 7) above.

Call Operator

The call operator transfers to the called procedure passing (in pointer register 0) a pointer to the argument list to be passed to the called procedure. This operator is invoked in two ways from ALM procedures. The first, as a result of the call pseudo-op, invokes the call operator after saving the machine registers in the calling program's stack frame. Upon return to the calling program, these saved values are restored into the hardware registers by the calling procedure. The second way that ALM procedures can invoke the call operator is as a result of the short_call pseudo-op. This is used rather than the call pseudo-op where the calling procedure does not need all of the machine registers saved and restored across the call. The ALM procedure can selectively save whatever registers are needed.

Note that neither the call nor the short_call pseudo-ops (nor the PL/I and FORTRAN equivalents) require or expect the machine registers to be restored by the called procedure. In

fact only the pointer registers 0 (operator segment pointer), 6 (stack frame pointer), and 7 (stack base pointer) are ever guaranteed to be restored across a call. It is up to the calling procedure to save and restore any other machine registers which are needed.

Entry Operator

The entry operator used by ALM programs performs two functions. It generates a pointer to the linkage section of the called procedure (which it leaves in pointer register 4) and it stores a pointer to the entry in what will be the stack frame of the called procedure (if the procedure ever creates a stack frame for itself). At the time the entry operator is invoked a new stack frame has not yet been established. Indeed, the called procedure may never create one. However, it is certainly possible to know where the stack frame will go if and when it is created and this knowledge is used to store the entry pointer.

The entry operator is invoked by an ALM procedure when a procedure transfers to a label in another procedure when has been declared as an entry via the entry pseudo-op. The transfer is actually made to an entry structure which consists of the following (ALM) code:

```
        vfd   18/entry_definition,18/0

entry:  tspbp sb|entry_op,*
```

The operator returns to the instruction after the tspbp instruction which may or may not be another transfer instruction. The word before the tspbp instruction contains a pointer (relative to the base of the definition section) to a definition for the entry. The definition will yield the entry name and (eventually) the number and type of arguments expected by the entry. Note that a link to the entry will, when snapped, point to the tspbp instruction. It is therefore an easy matter to find information about the entry given a pointer to the entry.

Note that some ALM programs may not require a linkage pointer. Such programs can declare the label to be transferred to with a segdef pseudo-op. This will cause the appropriate definition and linkage information to be generated so that other procedures can find the entry point. When called, the transfer will be straight to the code at the label and the normal entry structure will not be generated or used. No linkage pointer will be found nor will an entry pointer be saved. This technique is

recommended only where speed of execution is of utmost importance since it avoids calculation of useful (debugging) information.

Push Operator

The push operator used by ALM procedures is invoked as a result of the push (formerly save) pseudo-op which is used to create a stack frame for the called procedure. In addition to creating a stack frame, several pointers are saved in the new stack frame. These are:

1) argument pointer

2) linkage pointer

3) previous stack frame pointer

4) next stack frame pointer

If the called procedure were defined as an entry (rather than segdef) the entry pointer would already have been saved in the new stack frame.

The push pseudo-op must be invoked if the called procedure makes further calls itself or uses temporary storage. Note that due to their manner of execution, PL/I and FORTRAN procedures combine the entry and push operators into a single operator.

The push operator (and the return operators to be mentioned next) are managers of the stack frames and the stack segment in general. The push operator establishes the forward and backward stack frame threads and updates the stack end pointer in the stack header appropriately. The return operators use these threads and also update the stack end pointer as needed. Any program which wishes to duplicate these functions must do so in a way which is compatible with this document and the MPM Subsystem Writers' Guide section, Standard Stack and Linkage Area Formats.

Return Operator

The return operator is invoked by ALM procedures which have specified the return pseudo-op. The return operator pops the stack, reestablishes the minimum execution environment and returns control to the calling procedure. Note that the only registers restored are pointer registers 0, 6 and 7 as mentioned above and none of the arithmetic registers.

Short Return Operator

The short_return operator is invoked by ALM procedures which have specified the short_return pseudo-op. The short return operator differs from the return operator in that the stack frame is not popped. This return is used by ALM procedures which did not perform a push.

Code Sequences

The following code sequences are the ones that will be generated by the assembler for the specified pseudo-op. Note that the code generated by ALM is the same for all versions of the assembler and that only the code invoked (i.e., in the operator segment) changes as we progress to the final call/push/return strategy.

There are three sets of expanded code corresponding to the following three versions of the operators:

1) the operators installed on the 645 prior to changing over to the new call/push/return strategy,

2) the operators installed on the 645 after changing over to the new call/push/return strategy,

3) the operators that will be used on the Honeywell 6180 hardware.

The code in the left hand columns is what will be generated by the indicated pseudo-op. This code will be found in the actual object code. The code in the right hand columns is the code in the operator segment that will be invoked by the corresponding pseudo-op.

Interim Honeywell 645 Call/Push/Return Code Sequences

call:

```
        stb     sp|0
        sreg    sp|32
        eapap   arglist
        eapbp   entrypoint
        tsblp   sb|call_op,*
```

```
                              stplp    sp|return_ptr
                              sti      sp|return_ptr+1
                              eaplp    sp|lp_ptr,*
                              tra      bp|0

          ldb       sp|0
          lreg      sp|32

short_call:

          eapbp     entrypoint
          tsblp     sb|call_op,*

                              (as above)

          eaplp     sp|lp_ptr,*

return:

          tra       sb|return_op,*

                              ldb      sp|prev_sp,*
                              lreg     sp|8
                              rtcd     sp|return_ptr

short_return:

          tra       sb|short_return_op,*

                              ldb       sp|0
                              lreg      sp|8
                              rtcd      sp|return_ptr

entry:

          tsbbp     sb|entry_op,*

                              eapbp    bp|-1
                              eaplp    sp|next_sp,*
                              stpbp    lp|entry_ptr
                              lda      lp|entry_ptr
                              lda      sb|lot_ptr,*au
                              eablb    0,au
                              eablp    0,al
                              tra      bp|1

          tra       executable_code
```

push:

```
          eax7     stack_frame_size
          tsbbp    sb|push_op,*

                                stbpb    sp|next_sp,*
                                eapbp    sp|next_sp,*
                                stpsp    bp|prev_sp
                                stpap    bp|arg_ptr
                                stplp    bp|lp_ptr
                                eapsp    bp|0
                                eapbp    sp|0,7
                                stpbp    sp|next_sp
                                eapbp    sp|0,*
                                stb      sp|0
                                tra      bp|0
```

Final Honeywell 645 Call/Push/Return Code Sequences

call:

```
          stb      sp|0
          sreg     sp|32
          eapap    arglist
          eapbp    entrypoint
          tsblp    sb|call_op,*

                                stplp    sp|return_ptr
                                eaplp    sp|lp_ptr,*
                                tra      bp|0

          ldb      sp|0
          lreg     sp|32
```

short_call:

```
          eapbp    entrypoint
          tsblp    sb|call_op,*

                                (as above)

          eaplp    sp|lp_ptr,*
```

```
return:

          tra       sb|return_op,*

                              stpsp     sb|stack_end_ptr
                              eapsp     sp|prev_sp,*
                              eapap     sp|op_ptr,*
                              rtcd      sp|return_ptr

short_return:

          tra       sb|short_return_op,*

                              eapap     sp|op_ptr,*
                              rtcd      sp|return_ptr
entry:

          tsbbp     sb|entry_op,*

                              eapbp     bp|-1
                              eaplp     sb|stack_end_ptr,*
                              stpbp     lp|entry_ptr
                              lda       lp|entry_ptr
                              lda       sb|lot_ptr,*au
                              eablb     0,au
                              eablp     0,al
                              tra       bp|1

          tra       executable_code


push:

          eax7      stack_frame_size
          tsbbp     sb|push_op,*

                              stpbp     sb|stack_end_ptr,*
                              eapbp     sb|stack_end_ptr,*
                              stpsp     bp|prev_sp
                              stpap     bp|arg_ptr
                              stplp     bp|lp_ptr
                              eapsp     bp|0
                              eapbp     sp|0,7
                              stpbp     sb|stack_end_ptr
                              stpbp     sp|next_sp
                              tra       sp|0,*
```

Honeywell 6180 Call/Push/Return Code Sequences

call:

```
          spri      sp|0
          sreg      sp|32
          eppap     arglist
          eppbp     entrypoint
          tsplp     sb|call_op,*

                                spri1p    sp|return_ptr
                                epplp     sp|lp_ptr,*
                                callsp    bp|0

          lpri      sp|0
          lreg      sp|32
```

short_call:

```
          eppbp     entrypoint
          tsplp     sb|call_op,*

                                (as above)

          epplp     sp|lp_ptr,*
```

return:

```
          tra       sb|return_op,*

                                sprisp    sb|stack_end_ptr
                                eppsp     sp|prev_sp,*
                                epbpsb    sp|0
                                eppap     sp|op_ptr,*
                                rtcd      sp|return_ptr
```

short_return:

```
          tra       sb|short_return_op,*

                                epbpsb    sp|0
                                eppap     sp|op_ptr,*
                                rtcd      sp|return_ptr
```

```
entry:

          tspbp     sb|entry_op,*

                              eppbp     bp|-1
                              epplp     sb|stack_end_ptr,*
                              spribp    lp|entry_ptr
                              epaq      bp|0
                              lprplp    sb|lot_ptr,*au
                              tra       bp|1

          tra       executable_code

push:

          eax7      stack_frame_size
          tspbp     sb|push_op,*

                              spribp    sb|stack_end_ptr,*
                              eppbp     sb|stack_end_ptr,*
                              sprisp    bp|prev_sp
                              spriap    bp|arg_ptr
                              sprilp    bp|lp_ptr
                              eppsp     bp|0
                              eppbp     sp|0,7
                              spribp    sb|stack_end_ptr
                              spribp    sp|next_sp
                              tra       sp|0,*
```

Conventions

The following conventions are used in the standard environment and should be followed by any translators or code written by users.

1) The only registers that will be restored across a call are the pointer registers

   0 (ap) operator segment pointer

   6 (sp) stack frame pointer

   7 (sb) stack base pointer.

   The operator segment pointer will be restored correctly only if it is saved at some time prior to the call (e.g., at entry time).

2) The code generated by the ALM assembler assumes that pointer register 4 (lp) always points to the linkage section for the executing procedure.

3) While operating on the 645 the bases are paired in the normal manner (i.e., even bases are internal, and the pairings are 0-1, 2-3, 4-5, 6-7).

The Argument List

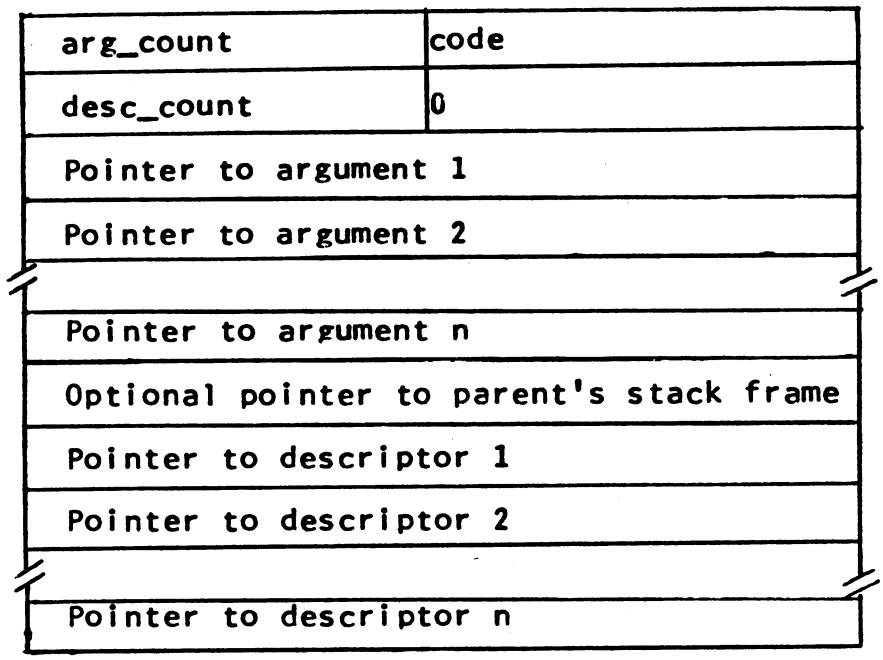
When a standard call is performed, the argument pointer (pointer register 0) is set to point at the argument list to be used by the called procedure. The argument list is a sequence of pointers and control information about the arguments. The argument header contains a count of the number of arguments, a count of the number of descriptors and a code specifying whether the argument list contains an extra stack frame pointer. The format of the argument list is shown below. In this diagram, n is the number of arguments passed to the called procedure. The following descriptions refer to items appearing in the diagram below.

arg_count — is in the left half of word 0, and is two times the number of arguments passed.

code — is in the right half of word 0, and is 4 for normal intersegment calls and 10 (octal) for calling sequences that contain an extra stack frame pointer. This pointer occupies the two words following the last argument pointer. It is present for calls to PL/I internal procedures and for calls made through PL/I entry variables.

desc_count — is in the left half of word 1 and is two times the number of descriptors passed. If this number is non-zero it must be two times the number of arguments passed; i.e., if any descriptors are passed, all must be passed.

The argument list must begin on an even word boundary. The pointers in the argument list need not be ITS pointers; however, they must be pointers which the hardware can indirect through. Packed (unaligned) pointers cannot be used. The format of an argument list follows:



The i'th argument pointer points to the i'th argument directly. The i'th descriptor pointer points to the descriptor associated with the i'th argument. The format for a descriptor is as follows:

```
declare 1 descriptor,
          2 flag bit(1),
          2 type bit(6),
          2 packed bit(1),
          2 number_dims bit(4),
          2 size bit(24);
```

1) flag — always has the value "1"b and is used to tell this descriptor format from an earlier format.

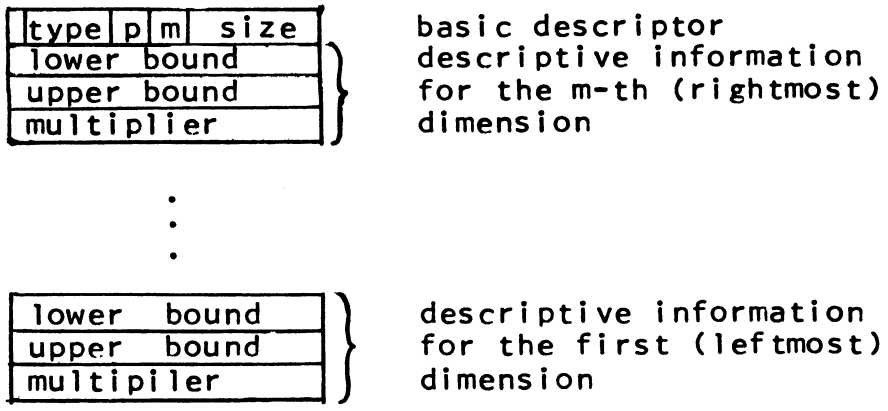
2) type — is the data type according to the following encoding:

1 Real Fixed Binary Short
2 Real Fixed Binary Long
3 Real Floating Binary Short
4 Real Floating Binary Long
5 Complex Fixed Binary Short
6 Complex Fixed Binary Long
7 Complex Floating Binary Short
8 Complex Floating Binary Long
9 Real Fixed Decimal

10 Real Floating Decimal
11 Complex Fixed Decimal
12 Complex Floating Decimal
13 Pointer
14 Offset
15 Label
16 Entry
17 Structure
18 Area
19 Bit-string
20 Varying Bit-string
21 Character-string
22 Varying Character-string
23 File

3) packed — has the value "1"b if the data item is packed. (Shown as "p" in the diagram of a descriptor below.)

4) number_dims — holds the number of dimensions in an array. (Shown as "m" in the diagram of a descriptor below.) The array bounds and multipliers follow the base descriptors as follows:



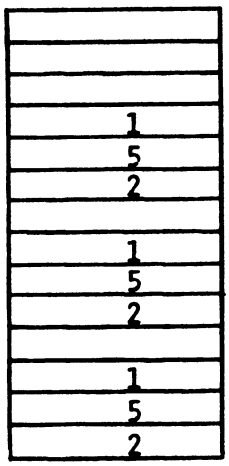
If the data is packed the multipliers give the element separation in bits, otherwise they give the element separation in words.

5) size — holds the size (in bits, characters or words) of string or area data, the number of structure elements for structure data, or the scale and precision (as two 12-bit fields) for arithmetic data. For arithmetic data the scale is recorded

in the leftmost 12 bits and the precision is recorded in the rightmost 12 bits. The scale is a 2's complement, signed value.

The descriptor of a structure is immediately followed by descriptors of each of its members. For example, (assuming that each element of C or D occupies one word) the following declaration produces the descriptor shown.

```
declare 1 S,
          2 A,
          2 B (5),
            3 C,
            3 D;
```



	basic descriptor of S
	basic descriptor of A
	basic descriptor of B
1	lower bound of B
5	upper bound of B
2	element separation of B
	basic descriptor of C
1	lower bound of C
5	upper bound of C
2	element separation of C
	basic descriptor of C
1	lower bound of D
5	upper bound of D
2	element separation of D

Note that members of dimensioned structures are arrays and their descriptor contains copies of the bounds of the containing structure.

INTRAPROCESS ACCESS CONTROL (RINGS)

Control of access between processes is described in the MPM Reference Guide section, Access Control. However, the ability to grant distinct access rights on segments only to different processes is not sufficient control in Multics. For example, those procedures and data segments which control the performance of critical system functions (e.g., process switching, paging, access control, etc.) within a process must be protected from tampering by user programs. This type of intraprocess access control is achieved by the Multics ring mechanism. Actually, the ring mechanism is a generalization of the simple supervisor/user protection scheme required by the example above. The ring mechanism permits users to write subsystems which are protected from other users in much the same way the supervisor is protected from users. For example, consider a user who wished to have a segment contain a list of his employees and their salaries. He wants to permit other users to have access to the list of employees and he will allow some other users to know the average salary of his employees, but he does not want anyone else to have access to the specific salary of each employee. In other words he wishes other users to have access to the data in the segment but only in a controlled manner and he wants to be able to specify the control. The read, write, and execute access modes of segments do not provide this capability; however, the ring mechanism does. In effect, rings permit arbitrarily refined and controlled access to objects by allowing any user to define arbitrary objects and write procedures which operate on these objects, and encapsulate these procedure and objects in a closed controlled environment which can be entered at specific entry points.

One of the important properties of the ring mechanism is that it is invisible to users of segments that are protected by it. Only those programmers actually writing subsystems that need ring protection need be familiar with the use and operation of rings. All other users need no further information about access control in the storage system and need not read the remainder of this section.

Conceptually, the Multics ring mechanism is quite straightforward. Picture a series of concentric circles. Let all the segments in a process reside somewhere in the picture, such that each segment is between the boundaries of some pair of circles, or in the innermost circle. Now label the areas contained by the circles, starting with the innermost, from R0 to R7. (The "R" stands for "ring", for reasons which are probably obvious.) The primary rule is that segments in the same ring have free access to one another, subject to any limitations

prescribed by their access modes. Access between rings is limited according to rules given below. The first point to notice, however, is that once we have established the ring model, we provide for walling off ordinary user segments from those segments which belong to the supervisor by assigning the segments to different rings.

The primary rule of access between rings is that segments in lower-numbered rings have, in general, unlimited access to segments in higher-numbered rings, subject, of course, to mode restriction on particular segments; whereas segments in higher-numbered rings have no direct access to segments in lower-numbered rings. Access refers to both the ability to execute a segment and the ability to read or write it. Thus, from the outside of the ring structure looking in toward the central supervisor in ring 0, the ring boundaries are walls. Recall that within a ring (which is to say, between walls) life goes on unimpeded by the protection mechanism. It is when a wall must be crossed that the protection mechanism comes into play.

Those segments which comprise the central supervisor are in ring 0. Ring 1 contains system routines, largely administrative in nature, which are not so sensitive as the central supervisor and which cause less disastrous results in case of failure. Rings 3 through 7 are potentially available for use by users. Most user processess start running in ring 4. A built-in advantage of this structure is that users may avail themselves of "spheres of protection" just as the supervisor does.

In a process, the ring which contains the currently executing segment is called the current ring of execution and is part of the state of the process. Control must, of course, be able to pass from ring to ring. By virtue of the ring structure's basic definitions, passing control outward is legal. That is, segments in outer rings are accessible to those in inner rings. However, by virtue of those same definitions we have yet to see a way in which an inward call could be legal. That is, segments in inner rings are in general inaccessible to those in outer rings. The means of legitimizing inward calls is to cause one or more entry points of a given procedure segment to be treated as gates in the protection wall. A gate, then, is an entry point to an inner ring procedure segment which may be called by an outer ring segment.

Segment Ring Brackets

The description given so far indicates that each segment in the system must be a member of a single ring and if the segment

is executable, it executes in that ring. It is, however, very often convenient to allow a segment to reside in (be a member of) several rings so that it may execute in any of these rings with the access appropriate to that ring. This is accomplished by giving each segment an execute bracket that delimits the rings in which the segment may be executed if it has execute access mode without having to change the ring of execution of the process. The execute bracket is specified by means of two ring numbers, for example, 3, 5. The execute bracket includes all rings between and including the two ring numbers; in the example the execute bracket contains rings 3, 4, and 5. If the process is executing in a ring contained in the execute bracket of a segment and control is transferred to the segment then no change of ring of execution results. If the process is executing in a ring whose number is less than the lowest ring number in the execute bracket, then when the segment is transferred to, the ring of execution will be changed to the lowest ring in the execute bracket. In the example, if the process is executing in ring 1 and the segment is transferred to, then the ring of execution will become 3. If the process is executing in a ring whose number is greater than the highest ring number in the execute bracket, then the ring of execution will become the highest ring in the execute bracket (5 in the example) when the segment is transferred to, assuming the segment contains a gate. In this latter case of gates it is also useful to specify those rings in which the segment is accessible through a gate. This gate bracket is specified by appending a third ring bracket number after the two already used for defining the execute bracket, e.g., 3, 5, 6. The gate bracket includes those rings whose number is greater than the second ring bracket number and less than or equal to the third bracket number (in the example only ring 6). An attempt to execute a segment from a ring greater than the gate bracket is not allowed. The execute bracket is still specified by the first two ring bracket numbers as before.

Since, in order to contain a gate, a segment must have a non-empty gate bracket, it is convenient to choose a non-empty gate bracket for a segment as the definition of a gate segment; e.g., an executable segment with ring bracket numbers 4, 5, 7 is a gate segment because its gate bracket contains rings 6 and 7, whereas an executable segment with ring bracket numbers 4, 5, 5 is not a gate segment because its gate bracket is empty. In the current implementation, gate segments must also have a specific format. See the MPM Subsystem Writers' Guide section on translate_gate for a description of how to generate gate segments. The above use of ring bracket numbers dictates that they be increasing; i.e., the first ring bracket number must be

less than or equal to the second ring bracket number which, in turn, must be less than or equal to the third ring bracket number.

The ring bracket numbers also have meaning with respect to the read and write access modes. The rings less than or equal to the first of the ring bracket numbers are termed the write bracket. A process must be executing in a ring within the write bracket of a segment and have write mode on that segment in order to write the segment. If a process is running in a ring higher than the write bracket then it may not write into the segment even though the process has write mode on the segment. The rings less than or equal to the second ring bracket number are called the read bracket. Processes must be running in the read bracket of a segment and have read mode to the segment in order to read it. The table below summarizes the refinements of access which are controlled by the ring brackets of a segment and the process's ring of execution, assuming the process has read, write and execute access modes specified on the ACL of the segment. Remember that ring brackets do not grant access to a segment by a process. Access to a segment by a process is granted only by the Access Control List of a segment. Ring brackets only serve to refine, within the process, the access modes granted by the Access Control List.

Ring of Execution	Potential Access Rights
Ring of execution less than first ring bracket number	read, write, execute (with ring change)
Ring of execution equal to first ring bracket number	read, write, execute
Ring of execution greater than first ring bracket number and less than or equal to second ring bracket number	read, execute
Ring of execution greater than second ring bracket number and less than or equal to third ring bracket number	execute (if a gate only, with ring change)
Ring of execution greater than third ring bracket number	no access

Validation Level

Inner ring procedures are very often called by outer ring procedures in order to perform some service on behalf of the outer ring. It is, therefore, necessary that the inner ring procedure know the number of the outer ring on whose behalf it is performing the service in order that it may validate that the outer ring has the right to request the service. This requesting ring information is kept by each process and is known as the validation level. If an outer ring procedure wishes to request a service from an inner it simply sets the validation level to its current ring of execution (the validation level may not be set lower than the ring of execution) and calls the inner ring procedure. If a procedure is calling an inner ring procedure to do work on behalf of an outer ring procedure then it should not change the validation level, but instead leave it at the level of the outer ring procedure. Note that users who write programs which are executed only in a single ring, which is usually the outermost ring in which the process runs, need not be concerned about the validation level since it will be set to that ring by default.

Directory Ring Brackets

Directory ring brackets are, in most ways, similar to segment ring brackets. There is, however, one important difference. Since directories are accessed by calling supervisor primitives rather than by direct hardware reference, the directory ring brackets are evaluated with respect to the validation level instead of the ring of execution. There are two ring bracket numbers associated with each directory. The first ring bracket number defines the modify/append bracket. All rings less than or equal to the first directory ring bracket number are within the modify/append bracket. In order for a process to modify a directory or add entries to a directory, the validation level of the process must be within the modify/append bracket and the process must have modify or append access modes. The rings less than or equal to the second directory ring bracket number form the status bracket. In order to get the attributes of segments or directories within a directory, the validation level must be within the status bracket. The first ring bracket number must be less than or equal to the second ring bracket number. For example, if the ring brackets of a directory are 4, 6 then if the validation level is 3, the process may get status of, modify, or append to the directory (assuming, of course, that it has the status, modify, and append modes). If the validation level is 6, it may only get status of the directory. If the validation level

is 7, it may not access the attributes of the entries in the directory at all.

The ring brackets of segments or directories may be modified by using the set_ring_brackets or set_dir_ring_brackets commands or the hcs_$set_ring_brackets or hcs_$set_dir_ring_brackets subroutines. (See the MPM Subsystem Writers' Guide write-ups for these commands and subroutines.*)

The Modification of Segment Attributes

In order to maintain the integrity of the ring mechanism, it is necessary that the ring brackets of a segment or directory control a process's ability to modify the attributes (particularly the ACL and ring brackets) of a segment as well as its ability to write the data of the segment. As stated previously, in order to modify the attributes of a segment or directory within a directory the process must have modify access to the latter directory and the validation level must be within the modify/append bracket of the directory. This is refined by the further qualification that the validation level be within the write bracket of a segment whose attributes are being modified or be within the modify bracket of a directory whose attributes are being modified. Also, a process may not set bracket numbers to values less than the validation level. Finally, to prevent one protected subsystem from tampering with another protected subsystem in the same ring, an ACL entry containing a project identifier other than the project of the executing process may not be added to the ACL of a gate segment. These restrictions insure that there are no means by which a process executing in a ring outside the write bracket may directly write a segment or do so indirectly by first modifying the ring brackets or ACL of the segment to give the process write access and then write it. This assures the integrity of the ring mechanism.

Since the final type of directory entry, the link, has no access control of its own, it is simply necessary to be able to modify its attributes. To do this the process must simply have modify mode on the directory containing the link and the validation level must be within the modify bracket of the directory.

Default Values

When a segment or directory is created and the values for the ring bracket numbers are not explicitly defined they will be set to a default value equal to the validation level. Since most users write programs that operate in a single ring at a single

* Some are not yet documented as of January 10, 1973.

validation level, this choice of default ring brackets makes the ring mechanism invisible to them, i.e., they will always be within the read, write, execute, status, and modify brackets.

As explained in the MPM Reference Guide section, Access Control, when a new segment or directory is appended to a directory, the default value for the access control list is determined by the initial ACLs. In each directory there is a set of initial ACLs for newly created segments and a set of initial ACLs for newly created directories. The reason why a set of initial ACLs rather than simply a single initial ACL is necessary is now explained. The set contains one initial ACL for each ring. When a segment or directory is created the initial ACL corresponding to the validation level is the one used. Since the initial ACL for a given ring can be modified only by procedures in rings equal to or less than the given ring, a procedure creating a new segment or directory can be sure that the initial ACL to be used could not have been modified by a ring less privileged than the ring on whose behalf the segment or directory is being created.

References

Discussion of the implementation of protection rings may be found in:

1) Organick, E.I., The Multics System: An Examination of Its Structure, Chapter 4, Access Control and Protection, M.I.T. Press, Cambridge, Mass., 1972.

2) Schroeder, M.D., and Saltzer, J.H., "A Hardware Architecture for Implementing Protection Rings", ACM Third Symposium on Operating System Principles (October 18-20, 1971), Palo Alto, California.

3) The Multics Virtual Memory, AG95, Rev. 0, File No. 1J12, Honeywell Information Systems, Inc., 1972

(a collection of 3 technical papers -

. "The Multics Virtual Memory", TIS Report R69LSD3, Copyright 1970 by General Electric Co., U.S.A.

. "Access Control to the Multics Virtual Memory", TIS Report R69LSD4, Copyright 1970 by General Electric Co, U.S.A.

. "Series 6000 Features for the Multics Virtual Memory").

 (END)

HARDWARE AND SIMULATED FAULT ASSIGNMENTS

Faults in Multics are initially handled by Hardcore Ring procedures. Many are then signalled as conditions. For the default handling of these conditions see the MPM Reference Guide section, List of System Conditions and Default Handlers.

Hardware Faults

Most hardware faults occurring in Multics are reserved for special interpretation by the system. Two are reserved for users. The following list of Multics hardware faults describes the interpretation given to each fault, or notes that it is reserved by the system for future assignment. The list is ordered by the conditions which are signalled as a result of the faults. Note that in some cases the description of the fault is not exhaustive, but rather documents the most likely situations. If the user encounters a situation which appears not to be covered here, he should consult the H645 processor manual.

accessviolation (Access Violation Subcondition of the Illegal Procedure Fault)
The user attempted to access a segment in a manner not permitted by the user's access mode to that segment.

derail (Derail Fault)
The DRL machine instruction was encountered. This fault is reserved for users.

execute_data (Execute Data Subcondition of the Illegal Procedure Fault)
The user attempted to execute data. The fault is used to intercept attempted outward wall crossings.

fault_tag_1, fault_tag_3 (Fault Tag 1, Fault Tag 3)
A fault tag 1 or fault tag 3 tally appeared in the address modifier field of an indirect word. Fault tag 1 is equivalent to fault tag on the H635. It is used by the BASIC system, but is otherwise reserved for users; fault tag 3 is reserved for future assignment.

gate_error (Directed Fault 2)
A directed fault 2 code appeared in a Segment Descriptor Word (SDW). The Multics system interprets it as an attempted inward wall crossing and signals the gate_error condition in the event of an error, as noted in the MPM Reference Guide section, List of System Conditions and Default Handlers.

Hardware and Simulated Fault Assignments
Multics Programming Environment
Page 2

illegal_descriptor (Illegal Descriptor Fault)
An illegal Segment Descriptor Word (SDW) or Page Table Word (PTW) was referenced.

illegal_memory_command (Illegal Memory Command Fault)
A memory controller received an inappropriate request (e.g., a read clock command on a memory controller not containing a clock); or a processor issued a connect to a masked channel.

illegal_opcode (Illegal Opcode Subcondition of the Illegal Procedure Fault)
The user attempted to execute an illegal operation code.

illegal_procedure (Illegal Procedure Fault)
The user attempted a programming violation which could affect other users on the system. See accessviolation, execute_data, illegal_opcode, and out_bounds_err.

linkage_error (Fault Tag 2)
A fault tag 2 tally appeared in the address modifier field of an indirect word. Multics interprets this as a linkage fault and signals linkage error if the intersegment link reference cannot be successfully resolved.

mme1, mme2, mme3, mme4 (Master Mode Entry 1 Fault, Master Mode Entry 2 Fault, Master Mode Entry 3 Fault, Master Mode Entry 4 Fault)
A MME1 (2, 3, 4) machine instruction was encountered. MME1 on the H645 is equivalent to MME on the H635. The MME1 instruction is used within the Dartmouth system, MME2 is used by the debug command, and MME3 and MME4 are reserved for future assigment.

op_not_complete (Operation Not Complete Fault)
A machine operation was not completed in the required time; or H645 programming rules were violated; or an ITS or ITB modifier appeared in an odd location; or a processor addressed a system controller not attached to it.

out_bounds_err (Out of Bounds Subcondition of the Illegal Procedure Fault)
The user attempted to reference a nonexistent location, either by a segment offset beyond the end of the segment specified or by a segment number not known to the process. If the segment number is -1 (in 2's complement form), a simfault_nnnnnn will be signalled instead. (See Simulated Faults below.)

ovrflo (Overflow Fault)
An arithmetic operation exceeded the precision of the variables involved.

page_fault_error (Directed Fault 1)
A directed fault 1 code appeared in a Page Table Word (PTW). The Multics system interprets it as an indication that a page is not in memory, and signals the page_fault_error condition if a referenced page cannot be brought into memory.

parity (Parity Fault)
A memory location was referenced which has incorrect parity.

record_quota_overflow (Directed Fault 1)
A directed fault 1 code appeared in a Page Table Word (see page_fault_error above) for a page which did not yet exist either in memory or on secondary storage. If an attempt to create the page would cause the user's quota of secondary storage records to be exceeded, then the record_quota_overflow condition is signalled.

seg_fault_error (Directed Fault 0)
A directed fault 0 code appeared in a Segment Descriptor Word (SDW). The Multics system uses this as an indication that a segment is either not active or needs to have its access recomputed. The seg_fault_error condition is signalled if the segment does not exist, the user has incorrect access to the segment, or the segment number used is invalid.

undefined_acc (Directed Fault 4)
A directed fault 4 code appeared in a Segment Descriptor Word (SDW). The Multics system uses this as an indication that a segment has an illegal access combination.

zerodivide (Divide Check Fault)
The user attempted to divide by zero.

635_compatibility (635 Compatibility Fault)
The user attempted to execute an H635 instruction which does not exist on the H645.

635/645_compatibility (635/645 Compatibility Fault)
The user attempted to execute an instruction which is privileged in both H635 and H645 repertoires.

Simulated Faults

By convention the segment number -1 (in 2's complement form) is reserved for software simulated faults. The segment number is a dummy; i.e., no Multics segment will ever have it. Any attempt to reference that segment number will result in the out of bounds subcondition of the illegal procedure fault. When this fault occurs, the fault interceptor will signal (in the ring where the fault occurred) the simfault_nnnnnn condition, where nnnnnn is the offset portion of the segment address that caused the fault. This convention provides an additional 256K faults, the first 128K of which are reserved for system use. The remaining 128K faults are available for user programs.

One of the software simulated faults reserved for system use is currently assigned. An offset of 1 (simfault_000001) is defined as the null pointer value for the PL/I pointer data type. Thus the null pointer has the value (in 2's complement form) of -1|1. It is useful to note here that an inadvertent reference by a user to a null pointer may not produce an address with an offset of 1. In many cases the null pointer will be modified by an incremental offset. Thus, a null pointer modified by an offset of 22 (octal) would produce the condition simfault_000023. Users who receive a message on their terminal indicating that a simfault occurred should check for inadvertent use of a null pointer.

Process Termination Fault

By convention the segment number -2 (in 2's complement form) is reserved for the process termination fault. Any reference to that segment number will cause the referencing process to be terminated. The offset portion of the segment address may be used to indicate the reason for the termination. Of the 256K possible offsets, the last 128K are reserved for interpretation by the system. The first 128K are available for user programs. Any offset currently recognized by the system will be interpreted in a message printed on the user's terminal after the process is terminated.

 (END)

LIST OF COMMAND CONTROL ARGUMENTS

Many Multics commands have control character strings as one or more of their arguments. These character strings have the hyphen (-) as their first letter. They are typically optional, with some default action to be taken if a particular control argument is not present. For example, the command line

```
copy old_seg new_seg
```

attempts to perform the specified segment copy and prints full comments when an error occurs, whereas

```
copy old_seg new_seg -brief
```

attempts to perform the same segment copy but does not comment on certain types of errors.

A list of the control arguments currently used on Multics follows. A command writer should consult this list before inventing a control character string, to see if any of these will serve his purpose. If not, he should notify the editor of the MPM that this section requires updating. The abbreviation for each control argument is shown when one exists; otherwise, "none" appears in the right-hand column.

Control Argument	Abbreviation
-7punch	-7p
-absentee	-as
-account	-ac
-acl	none
-admin	-am
-all	-a
-arguments	-ag
-author	-at
-ball	-bl
-bottom_up	-bu
-branch	-br
-brief	-bf
-call	-cl
-character	-ch
-check	-ck
-compile	none
-console_input	-ci
-copy	-cp
-date	-dt
-date_time_modified	-dtm
-date_time_used	-dtu

-debug	-db
-delete	-dl
-depth	-dh
-device	-dv
-directory	-dr
-entry	-et
-every	-ev
-file_input	-fi
-first	-ft
-force	none
-from	-fm
-halt	-ht
-header	-he
-hold	none
-home_dir	-hd
-hyphenate	-hph
-indent	-in
-last	-lt
-length	-ln
-library	-lb
-limit	-li
-link	-lk
-list	-ls
-long	-lg
-map	none
-mcc	none
-mode	-md
-multisegment_file	-msf
-name	-nm
-no_pagination	-npgn
-no_preempt	-np
-no_print_off	-npf
-no_restore	-nr
-no_start_up	-ns
-no_update	-nud
-number	-nb
-optimize	-ot
-output_file	-of
-queue	-q
-page	-pg
-parameter	-pm
-pass	none
-pathname	-pn
-print_off	-pf
-process_overseer	-po
-project	-pj
-raw	-none
-reset	-rs

-restart	-rt
-reverse	-rv
-ring_brackets	-rb
-segment	-sm
-severity_i	-sv_i
-source	-sc
-stop	-sp
-subscriptrange	-subrg
-symbols	-sb
-table	-tb
-time	-tm
-times	none
-to	none
-total	-tt
-type	-tp
-wait	-wt
-working_directory	-wd

 (END)

COMMANDS AND ACTIVE FUNCTIONS

This section contains, in alphabetic order, descriptions of the Multics commands and active functions primarily useful to subsystem writers. The following conventions are used in command descriptions:

1) In command usage, optional arguments are shown surrounded with hyphens. For example,

   locate name1 -name2-

   would indicate that the locate command has a mandatory first argument and an optional second argument.

2) In command usage, the ellipsis form

   a1 ... an

   is used to indicate a variable number of arguments all having the same form as a1 and an.

Note that commands may be distinguished from subroutines by name; in general, subroutines have segment names which end with a trailing underscore.

 (END)

Command
Development System
6/5/72

Name: error_table_compiler, etc

This command compiles a table of status codes and associated messages from symbolic ASCII source. The output is in a format suitable for the ALM assembler to produce a standard status code table.

Usage

error_table_compiler error_table

1) error_table specifies a source segment in the format noted below. A suffix of .et is added to complete the source segment name. The output segment is named error_table.alm. This segment must then be assembled by the ALM assembler prior to using it.

Notes

Each status code is defined by a statement in the source segment which specifies the entry name, short message, and long message associated with a status code. Each statement is terminated by a semicolon, and a colon is used to delimit the entry name. The short message must be less than nine characters and is terminated by a comma. The long message must be less then 101 characters and is terminated by the semicolon ending the statement. Any number of entry names may be given to a status code. These entry names must be 30 characters or less in length.

The syntax of a statement is:

<name>:[<name>: ...] <short message>, <long message>;

An error table source segment is composed of a series of statements of the above format, terminated by an end statement. The format of the end statement is

end;

There is a special statement which should not be used except when compiling the hardcore system error table. This statement causes a special nondynamic initialization of status codes in that segment, optimizing the system error table slightly. This statement may appear anywhere in the sources before the end statement. The format of this statement is:

system;

See also the MPM Reference Guide section List of System Status Codes and Meanings.

Example

Note the comment syntax, which is similar to PL/I, in the following example:

```
/*   This is a sample error table compiler source segment. */

too_few_arguments:    toofew,There were too few arguments.;

could_not_access_data:  noprivlg,The user is not sufficiently
privileged to access required data;

fatal: disaster:      disaster,There was a disastrous error in the
data base;

end;
```

Each status code in the table produced by error_table_compiler should be referenced as a fixed bin(35) quantity, known externally:

```
declare user_errors$disaster fixed bin(35) external,
        code fixed bin(35);

call data_base_manager (info, code);
if code = user_errors$disaster /* this is bad */
        then call kill_subsystem;
```

 (END)*

Name: make_commands, mc

This command creates a segment in a specific format from an ASCII input segment. This segment is referenced by the Limited Service System when it is limiting the commands and percentage of CPU time of a user.

The input segment consists of a series of statements. Each statement is composed of two parts. The first part is the name of the command to be transformed; i.e., the command that is to be typed by the user in a limited system. If there is more than one name for the command, they should all be enclosed in parentheses and separated from each other by one or more blanks. The name field is terminated by a colon preceded by any number of blanks.

The second part of each statement is the path name (which may be a relative path name) of the command to be executed when the user types one of the names in the first part. It is followed by any number of blanks and terminated by a semicolon. If the path name is omitted (semicolon still required), it is assumed to be the same as the last name in the name field.

The first and second parts of each statement may be separated from each other by any number of blanks or tabs. New lines are ignored and are allowed anywhere. Comments enclosed in "/*" and "*/" are allowed and are treated as blanks.

If the first two statements have as their first part the names "ratio" and "interval", respectively, the second parts of the two statements are assumed to be decimal integers to be assigned to the ratio and interval_length variables of lss_command_list_. Otherwise, the two variables are set to zero.

Usage

make_commands path_name

1) path_name is the path name of an ASCII input segment which has the name path_name.ct. The output segment will be named path_name and will be placed in the working directory.

Example of an Input Segment

```
/* set the ratio and interval */
ratio:      60;
interval:   120;
/* define commands */
(list ls):          >abc>special$list;
logout:     ;
edit:       bsys;
start:      ;
hold:       ;
(pr print) :        ;
```

 (END)

Name: set_max_length, sml

This command allows the max length of a nondirectory segment to be specified. The max length is the length beyond which the segment will not grow. Currently, max length is accurate to the system page size (1024 decimal words).

Usage

set_max_length path length -control_arg1- ... -control_argn-

1) path — is the path name of the segment whose max length is to be set. If the argument path is a link, the max length of the segment linked to will be set. The star convention may be used.

2) length — is the new max length expressed in words. If this length is not a multiple of the system page size, it will be converted to the next higher multiple of the system page size. The default radix is decimal.

3) control_argi — may be chosen from the following list of control arguments and may appear in any position:

-decimal, -dc — The max length is expressed in decimal words.

-octal, -oc — The max length is expressed in octal words.

-brief, -bf — suppresses a warning message that the length argument has been converted to the next multiple of the system page size.

Notes

If the new max length is less than the current length of the segment, the user will be asked if the segment should be truncated to the max length. If the truncation takes place, the bit count of the segment will be set also.

Currently, if the user has append (a) permission with respect to the segment, the enforcement of the max length may not be done properly.

The user must have modify access on the directory containing the segment in order to change its max length.

Examples

```
set_max_length mailbox -oc 10000
```

will set the max length of the segment mailbox in the working directory to 4 pages.

```
set_max_length *.archive 16000
```

will set the max length of all two-component segments with a second component of archive in the working directory to 16 pages.

 (END)

Name: set_ring_brackets, srb

This command allows a user to modify the ring brackets of a specified nondirectory segment. Since there is only one set of ring brackets per segment, ring brackets of all users appearing on the access control list (ACL) of the specified segment will be modified. The star convention may be used in the entry name of the specified segment.

Usage

```
set_ring_brackets path -rb1- -rb2- -rb3-
```

1) path is the relative path name of the segment whose ring brackets are to be modified.

2) rb1 is the number to be used as the first ring bracket of the segment. See Notes below.

3) rb2 is the number to be used as the second ring bracket of the segment. See Notes below.

4) rb3 is the number to be used as the third ring bracket of the segment. See Notes below.

Notes

If rb3 is omitted, the third ring bracket will be set to rb2. If rb2 and rb3 are omitted, the ring brackets will be set to rb1. If rb1, rb2, and rb3 are omitted, they will be set to the user's current validation level. The ring brackets must be in the allowable range 0 through 7 and must have the ordering rb1 $\leq$ rb2 $\leq$ rb3.

 (END)

Active Function
Development System
3/29/72

Name: user

This active function returns various user parameters. The following parameters are described in the MPM write-up of the user active function: name, project, login_data, login_time, anonymous, secondary, absentee, term_id, term_type, cpu_secs, log_time, preemption_time, brief_bit, and protected.

Usage

[user arg]

arg may have one of the following values:

1) account	is the user account ID.
2) weight	is the user weight times 10.
3) login_word	is the user log in word. It may be either login or enter.
4) process_overseer	is the path name of the user's process overseer procedure.
5) device_channel	is the hardware channel associated with the user terminal, e.g., "tty192".
6) n_processes	is the number of processes created for the user since log in time. It is 1 plus the number of new_proc commands.
7) group	is the user's load control group.
8) abs_queue	if the user is an absentee user, this tells which queue he is running in; otherwise it is "interactive".
9) attributes	are the user's attributes, determined at log in time by user control. They are separated by commas and end with a semicolon. The legal attributes are: administrator anonymous brief dialok guaranteed_login multip

no_eo
no_primary
no_secondary
nobump
nolist
nostartup
preempting
vhomedir
vinitproc

(END)*

SUBROUTINES

This section contains, in alphabetic order, descriptions of the Multics subroutines primarily useful to subsystem writers.

The following conventions are used in subroutine descriptions:

1) An entry declaration, suitable for verbatim copying into a calling program, is provided. Using such a declaration is recommended practice, since it helps reduce errors.

2) Calling sequences are normally given for the PL/I language. Users of other languages should translate the sequences accordingly.

3) Following the description of each argument, the notation (Input) or (Output) indicates that the argument is passed to or comes from the subroutine, respectively.

Note that subroutines can be distinguished from commands by name; generally, subroutines have names which end with a trailing underscore.

Some subroutine write-ups indicate that the status code argument should be declared fixed binary or fixed binary (17), which is an older standard. Although any of the three declarations will generally work correctly, fixed binary (35) should be used.

 (END)

Subroutine Call
1/31/73

Name: alloc_

The alloc_ subroutine is called by user programs for dynamic storage management. It allocates a contiguous block of words of a specified size in a specified area and returns a pointer to that block.

Usage

```
declare alloc_ entry (fixed bin(26), ptr, ptr);

call alloc_ (size, area_ptr, return_ptr);
```

1) size is the amount of storage to be allocated, in words. (Input)

2) area_ptr is a pointer to the base of the area in which the storage is to be allocated. (Input)

3) return_ptr is a pointer to the first data word in the allocated block. This first data word will be on an even word boundary. The pointer will be null on return if there is no more room in the area. (Output)

Notes

The amount of storage actually allocated will be 2**n , where n is large enough to contain size words plus 2 overhead words for block information. If no blocks of the optimum size exist in the area, alloc_ will break up larger blocks until one of the optimum size is obtained.

If alloc_ is unable to return a pointer to a block of the size desired, it will signal the area condition. On a return from the area condition handler, the allocation will be retried. Thus any area condition handler established by the user should free storage in the area to allow alloc_ to perform the allocation. Otherwise an infinite loop will result since alloc_ will continue to signal the area condition and retry the allocation.

Programs using the PL/I area attribute and the PL/I allocate and free statements should not explicitly call this subroutine since PL/I performs all the necessary manipulations.

Entry: alloc_$storage_

This entry is identical to alloc_ except that the storage condition rather than the area condition is signalled if the allocation cannot be made.

Usage

```
declare alloc_$storage_ (fixed bin(26), ptr, ptr);

call alloc_$storage_ (size, area_ptr, return_ptr);
```

Arguments are as above.

Note

The PL/I compiler will generate a call to alloc_$storage_ when the allocate statement is used with no area specified.

 (END)

Subroutine Call
1/31/73

Name: area_

This subroutine initializes a PL/I area at the location specified and of the length specified.

Usage

```
declare area_ entry (fixed bin(26), ptr);

call area_ (length, area_ptr);
```

1) length is the length in words of the area to be allocated. This length must include space for at least 24 words of overhead storage at the base of the area. The usable storage block must be exactly 2**n words, where n is 2 or greater. Thus the length argument must be at least 2**n + 24, and its minumum value is 28 (2**2 + 24). (Input)

2) area_ptr is a pointer to the base of the area, and must have an offset of 0 (mod 2). (Input)

Notes

If the length argument is less than 28, the area condition is signalled by area_.

Programs using the PL/I area attribute and the PL/I allocate and free statements should not explicitly call this subroutine since PL/I performs all the necessary manipulations.

 (END)

Subroutine Call
2/1/73

Name: area_assign_

This subroutine copies all of the contents of one area into another area, except the word which contains the size of the area. For the assignment of the contents of one area to another area to be successful, the receiving area must be large enough to contain all of the allocations in the sending area.

Usage

```
declare area_assign_ entry (ptr, ptr);

call area_assign_ (to_ptr, from_ptr);
```

1) to_ptr is a pointer to the receiving area. (Input)

2) from_ptr is a pointer to the sending area. (Input)

Notes

If area_assign_ is unable to complete the assignment because the sizes of the areas are incompatible (e.g., the sending area has too much allocated in it for the receiving area to hold), it signals the area condition.

Programs using the PL/I area attribute and the PL/I allocate and free statements should not explicitly call this subroutine since PL/I performs all the necessary manipulations.

 (END)

Subroutine Call
Development System
6/30/72

Name: condition_interpreter_

condition_interpreter_ can be used by subsystem condition handlers to obtain a formatted error message (the same message printed by default_error_handler_) for all conditions except quit, alrm, cput and special PL/I or FORTRAN conditions. (See the write-up for default_error_handler_ in the SWS.) Some conditions do not have messages and others cause special actions to be taken (such as ovrflo). These are described in Notes below.

Usage

```
declare condition_interpreter_ entry (ptr, ptr, fixed bin,
        fixed bin, ptr, char(*), ptr, ptr);

call condition_interpreter_ (areap, mptr, mlng, mode,
     mcptr, cond_name, wcptr, infop);
```

1) areap — is a pointer to the area in which the message is to be allocated, if the message is to be returned. For safety, the area size should be at least 250 words. If the message is to be printed, the pointer is null. (Input)

2) mptr — points to the allocated message if areap is not null; otherwise it is not set. (Output)

3) mlng — is the length of the allocated message if areap is not null. If areap is null, the length is not set. Certain conditions (see Notes below) have no messages; in these cases, mlng =0. (Output)

4) mode — is the desired mode of the message to be printed or returned. It is 0 for the current mode, 1 for the normal mode, 2 for the brief mode, and 3 for the long mode. (Input)

Arguments five through eight are defined and explained in the MPM Reference Guide section on The Multics Condition Mechanism. These are the first four arguments passed to condition handlers by the Multics signalling mechanism.

Notes

The following conditions cause a return with no message:

command_error
command_question

The ovrflo condition is remapped into

fixedoverflow
or
overflow
or
underflow

and the appropriate new condition is signalled. If the handler for the signalled condition returns, this procedure then returns with no message.

The following conditions, associated with the PL/1 I/O or math packages, are not recognized by condition_interpreter_:

area
conversion
endfile
endpage
error
key
name
record
size
storage
stringrange
stringsize
subscriptrange
transmit
undefinedfile

WARNING: This interface is likely to change.

(END)

Subroutine Call
1/9/73

Name: convert_status_code_

This entry point returns the short and long status messages from the data base error_table_ corresponding to a given system status code.

Usage

```
declare convert_status_code_ entry (fixed bin(35),
        char(8) aligned, char(100) aligned);

call convert_status_code_ (code, shortinfo, longinfo);
```

1) code is a status code as returned by system subroutines. (Input)

2) shortinfo is a short status message corresponding to code. (Output)

3) longinfo is a long status message corresponding to code; the message is padded on the right with blanks. (Output)

Notes

If code does not correspond to a legitimate status code, shortinfo will be "XXXXXXXX", and longinfo will be "Code ddd not found in error_table_", where ddd is the decimal representation of code. If code has a negative value, shortinfo will be "iostatus", and longinfo will be "IO Status oooooooooooo", where oooooooooooo is the 12-digit unsigned octal representation of the status code.

 (END)

Subroutine Call
Development System
6/30/72

Name: cu_

The procedure cu_contains a number of useful command utility programs, most of which are documented in the MPM. The functions provided are not directly available in the Pl/I language.

Entry: cu_$ready_proc

The ready_proc entry is used to call the process' current ready procedure. It takes an optional argument, which it passes to the ready procedure. The ready procedure is automatically invoked by the listener after each command line is processed. The ready procedure of the standard command environment prints the ready message.

Usage

```
declare cu_$ready_proc entry;

call cu_$ready_proc();
```

or

```
declare cu_$ready_proc entry (1 aligned, 2 bit(1) unaligned,
        2 bit(35) unaligned);

declare 1 mode aligned,
          2 ready_sw bit(1) unaligned,
          2 pad bit(35) unaligned;
```

1) mode.ready_sw is the ready switch. If it is "1"b, the ready procedure should print a ready message. Otherwise it should not. (Input)

2) mode.pad is reserved for future use and must be zero. (Input)

Note

If no argument is given, a static one is passed to the ready procedure. The default value of the static ready switch is "1"b. The value of the static ready switch may be obtained using cu_$get_ready_mode and changed using cu_$set_ready_mode (see below). The listener invokes cu_$ready_proc without an argument. The ready_off command turns off the static ready switch, the ready_on command turns it on, and the ready command calls cu_$ready_proc with an argument whose ready_sw component is "1"b. Thus if a user-written ready procedure honors the ready switch,

its printing of the ready message can be controlled by the standard ready, ready_on and ready_off commands.

Entry: cu_$get_ready_proc

This entry returns a pointer to the process' current ready procedure.

Usage

```
declare cu_$get_ready_proc entry (ptr);

call cu_$get_ready_proc (ready_ptr);
```

1) ready_ptr is a pointer to the current ready procedure. If it is null, then the standard system ready procedure is being used. (Output)

Entry: cu_$set_ready_proc

This entry allows the user to change his process' ready procedure.

Usage

```
declare cu_$set_ready_proc entry (ptr);

call cu_$set_ready_proc (ready_ptr);
```

1) ready_ptr is a pointer to the procedure entry point which is to become the process' new ready procedure. If ready_ptr = null, the standard system ready procedure will become the process' ready procedure. (Input)

Entry: cu_$get_ready_mode

This entry returns the value of the static ready mode.

Usage

```
declare cu_$get_ready_mode entry (1 aligned, 2 bit(1)
        unaligned, 2 bit(35) unaligned);

declare 1 mode aligned,
          2 ready_sw bit(1) unaligned,
          2 pad bit(35) unaligned;

call cu_$get_ready_mode (mode);
```

1) mode.ready_sw is the current value of the static ready switch. (Output)

2) mode.pad is reserved for future use. (Output)

Entry: cu_$set_ready_mode

The entry allows the user to change the value of the static ready mode.

Usage

```
declare cu_$set_ready_mode (1 aligned, 2 bit(1) unaligned,
        2 bit(35) unaligned);

declare 1 mode aligned,
          2 ready_sw bit(1) unaligned,
          2 pad bit(35) unaligned;

call cu_$set_ready_mode (mode);
```

1) mode.ready_sw is the new value of the static ready switch. (Input)

2) mode.pad is reserved for future use and must be zero. (Input)

(END)

Internal Interface
Administrative/User Ring
01/08/71

Name: dl_handler_

This procedure is used to resolve "moderr" file system errors for commands which attempt to delete segments. The user is asked whether he wants to delete a given file; if he does, dl_handler_ attempts to give him REWA access to the segment. The segment is not deleted; this is left up to the caller.

Entry: dl_handler_

Usage

```
declare dl_handler_ entry (char(*), char(168) aligned,
        char(32) aligned, fixed bin);

call dl_handler_ (id, dname, ename, code);
```

1) id is the name of the calling procedure. (Input)

2) dname is the pathname of the directory containing the segment whose mode is to be changed. (Input)

3) ename is the entry name of the segment whose mode is to be changed. (Input)

4) code is an error code:

=0 if the user wished to delete the file and the mode was changed;

=1 if the user did not want to delete the file;

=error_table_$moderr

if the user wished to delete the file and the mode could not be changed. (Output)

MESSAGE:

id: ename is protected - do you want to delete it?

(Leading blanks in response are ignored. Any response other than "yes" is interpreted as "no".)

Entry: dl_handler_$noquestion

This entry is the same as above except that no question is asked. That is, without asking, it attempts to change the mode of the given segment.

Usage

```
declare dl_handler_$noquestion entry (char(*), char(168)
        aligned, char(32) aligned, fixed bin);

call dl_handler_$noquestion (id, dname, ename, code);
```

1) id is ignored. (Input)

2-4) are the same as above.

Entry: dl_handler_$dblstar

This entry should be called when a command, which performs some sort of manipulation on segments, receives as the entry name portion of a pathname argument a name containing stars (e.g., "delete >udd>**", "unlink **", or "deletedir <cp>*.list"). Normally, this entry will be called with the entry name as "**". The user is asked if he really wants to do what he has typed and returns a code indicating his response. Note: the library routine check_star_ returns an error code of 1 if a name containing stars (except for "**") has been found and an error code of 2 if "**" has been found.

Usage

```
declare dl_handler_$dblstar entry (char(*), char(168)
        aligned, char(32) aligned, fixed bin);

call dl_handler_$dblstar (id, dname, ename, code);
```

1) id is the name of the calling procedure. (Input)

2) dname is the pathname of the directory for which the entry name applies. (Input)

3) ename is the entry name containing stars. (Input)

4) code =0 if the user wants to do what he typed;

=1 otherwise. (Output)

MESSAGE:

Do you want to 'id ename' in dname?

(e.g., "Do you want to 'unlink **' in >udd>m>cp?")

Entry: dl_handler_$dirdelete

This entry is similiar to $dblstar except that it is called when a command attempts to delete a directory. The user is asked whether he wants to delete a directory. The absolute pathname is used in asking this question.

Usage

```
declare dl_handler_$dirdelete entry (char(*), char(168)
        aligned, char(32) aligned, fixed bin);

call dl_handler_$dirdelete (id, dname, ename, code);
```

1) id — is the name of the calling procedure. (Input)

2) dname — is the pathname of the directory containing the directory to be deleted. (Input)

3) ename — is the entry name of the directory to be deleted. (Input)

4) code — =0 if the user wants to delete the directory;

=1 otherwise. (Output)

MESSAGE:

Do you want to delete the directory dname>ename?

(END)

Subroutine Call
4/30/73

Name: dprint_

This is the subroutine interface for the dprint and dpunch commands (see the command write-ups in the MPM Reference Guide). It causes a request to print or punch a segment to be added to the specified dprint queue.

Usage

```
declare dprint_ entry (char(*), char(*), ptr,
        fixed bin(35));

call dprint_ (dirname, ename, argp, code);
```

1) dirname is the path name of the directory containing the segment to be printed or punched. (Input)

2) ename is the entry name of the segment to be printed or punched. It may be the name of a link or a multi-segment file. (Input)

3) argp is a pointer to the argument structure described in Notes below. If no argument structure is supplied, argp should be null. (Input)

4) code is a standard system status code. (Output)

Notes

The dprint_ subroutine uses the structure described below to determine the details of the request. If no structure is supplied, default values will be used.

```
declare 1 dprint_arg based aligned,
          2 version fixed bin,
          2 copies fixed bin,
          2 delete fixed bin,
          2 queue fixed bin,
          2 pt_pch fixed bin,
          2 notify fixed bin,
          2 heading char(64),
          2 output_module fixed bin,
          2 dest char(12),

          /* limit of version 1 structure */

          2 carriage_control fixed bin,
          2 pghdr char(120),
          2 forms char(8),
```

```
        2 lmargin fixed bin,
        2 line_lth fixed bin,
        2 pad(3) fixed bin;
```

1) version — is the version number of the structure. It should be set to 1.

2) copies — is the number of copies requested. (The default is 1.)

3) delete — is 1 if the segment is to be deleted after printing or punching; otherwise it is zero. (The default is zero.)

4) queue — is the priority queue in which the request will be placed. (The default is 3.)

5) pt_pch — is 1 for a print request or 2 for a punch request. (The default is 1.)

6) notify — is 1 if the requestor is to be notified when the request has been completed; otherwise it is zero. This option is not implemented at present. (The default is zero)

7) heading — is the string to be used as a heading on the front page of the output. If it is a null string the requestor's name will be used. (The default is the null string.)

8) output_module — indicates the Device Interface Module (DIM) to be used in executing the request. 1 indicates printing, 2 indicates 7-punching, 3 indicates Multics Card Code (MCC) punching, 4 indicates "raw" punching. (The default is 1.)

9) dest — is the string to be used to indicate where the output should be delivered. If it is null, the requestor's project ID will be used. (The default is the null string.)

The remaining items of the structure are not used in the present version.

 (END)

Subroutine Call
4/30/73

Name: find_command_

The find_command_ procedure attempts to generate a pointer to the procedure entry point corresponding to a specified command name. The command name is a character string specifying the name of a command as it might appear in a Multics command line (i.e., it may contain >, <, and $ characters).

Usage

```
declare find_command_ entry (ptr, fixed bin, ptr,
        fixed bin(35));

call find_command_ (cptr, cl, eptr, code);
```

1) cptr is a pointer to the unaligned character string which contains the specified command name. (Input)

2) cl is the number of characters in the command name. (Input)

3) eptr is a pointer to the procedure entry point corresponding to the command name. (Output)

4) code is a status code. If code is zero, find_command_ was successful, otherwise an error condition has occurred. If an error condition is indicated, the caller may assumed that find_command_ has already printed an appropriate diagnostic on the stream "user_output". (Output)

Entry: find_command_$clear

An associative memory is maintained by which find_command_ remembers procedure entry pointers for the 16 most recently referenced command names. Whenever a procedure is removed from the address space of the process, this associative memory should be reset via a call to find_command_$clear. This call clears the entire associative memory.

Usage

```
declare find_command_$clear entry;

call find_command_$clear;
```

There are no arguments.

 (END)

Subroutine Call
1/31/73

Name: freen_

The freen_ subroutine returns a previously allocated block of storage in an allocation area to a list of free blocks of this size. It will attempt to combine blocks of the same size to produce one larger block.

Usage

```
declare freen_ entry (ptr);

call freen_ (block_pointer);
```

1) block_pointer is a pointer to the base of the block to be returned to free storage. (Input)

Notes

The block_pointer argument must have been originally returned to the user by alloc_ or alloc_$storage. (See the MPM Subsystem Writers' Guide section for alloc_.) If an improper pointer (i.e., one not pointing to the base of a block of previously allocated storage) is passed to freen_, unpredictable results may occur.

Programs using the PL/I area attribute and the PL/I allocate and free statements should not explicitly call this subroutine since PL/I performs all the necessary manipulations.

 (END)

Subroutine Call
2/7/73

Name: get_at_entry_

This subroutine returns the interface module type and the device id associated with a specified stream name. For a description of the attach table which maintains such information, set the MPM Reference Guide section, Use of the Input and Output System.

Usage

```
declare get_at_entry_ entry (char(*), char(*), char(*),
        char(*), fixed bin(35));

call get_at_entry_ (ioname, type, id, mode, code);
```

1) ioname is the stream name about which information is desired. (Input)

2) type is the type of attachment; i.e., the name of the associated interface module. (Output)

3) id is the identifier of the device or stream name to which the attachment has been made. (Output)

4) mode is not used at this time, and returns a null string (""). (Output)

5) code is a standard status code. (Output)

 (END)