MULTICS STAFF BULLETIN-27

FROM: Mike Spier

TO: MSB Distribution

DATE: March 6, 1972

SUBJECT: The new Standard Object Segment

This is the final iteration on the new Standard Object Segment format; it has been accepted by all principals and the intention is to immediately start on its implementation. The projected work will include modification of the new PL/1 compiler, the ALM assembler, the Binder, as well as all other language processor currently supported on Multics. Also, all object segment manipulating tools (such as decode_object_, linker and prelinker, debug etc.) will be appropriately modified to handle both current and new object segment formats in an upwards compatible way.

This new format is conditionally accepted as the new standard for object segments, however its final adoption will be delayed until it is practically implemented in ALM, V2PL1 and the Binder, and is demonstratively functional.

Recipients of this document are requested to read it and submit any comments or criticism within the shortest possible delay, so as to enable us to catch any bugs or inconsistencies before work progresses too far.

# THE MULTICS STANDARD OBJECT PROGRAM

This document presents a standard format for the Multics object program to assure its compatibility with the 'Multics machine', the implication being that a piece of code which successfully executes on the 645 processor is not necessarily a standard Multics object program, and that the concept of execution on the 'Multics machine' includes notions of pure recursive re-entrant procedure, access control, and such functions as dynamic linking, machine independent diagnostics and debugging, binding etc. This standard relates primarily to the external interfaces of an object program, the objective being to leave as much freedom of code generation as possible to the language processors, and to impose a certain discipline only in regards to code which interfaces with the external world. It is assumed that the reader is reasonably familiar with Multics.

Certain formats described within this document are identical to formats found in older non-standard object programs; others are new and therefore incompatible with older object programs. Such new formats are annotated, within this document, with the symbol (NEW) to allow the reader more ease in relating the present standard to older ones. Needless to say, all such new formats are upwards compatible and the standard service system tools are coded in such a way as to properly handle all officially recognized object program formats until such time when the present standard is applied to all object programs in the system.

The Multics standard object program is the only type of object program guaranteed to be supported by the Multics standard service system tools.

## Table of Contents

## 1. OVERVIEW

A Multics standard object program is an executable hardware-level (i.e., machine code) representation of some higher language algorithm, produced by the appropriate language processor. Physically, it is a single array of words at the base of a distinct segment known as an _object segment_.

The generated object code falls into several categories, the most important of which are,

_Text_ - the executable machine code representation of the desired algorithm.

_Definitions_ - symbolic information with the aid of which certain variables which are internal to the object program are made known to the external world and accessible to the dynamic linking mechanism (the Multics linker).

_Links_ - symbolic representation of variables whose address is unknown at compile time, and can only be evaluated (i.e., resolved into a machine address by the dynamic linking mechanism) at execution time.

_Symbol Tree_ - internal definition of symbolic source language variables, their attributes and relative address within the object segment; needed for the execution of interpretive code such as PL/1 Input/output as well as for debugging purposes.

_Historical Information_ - information describing the circumstances under which the object segment was created, such as name and version of language processor, creation time, identification of input source, identification of user who initiated the object segment creation, etc.

_Relocation Information_ - which identifies all instances of internal relative address references.

_Diagnostics Aids_ - information which allows standard system tools to extract useful information out of an object segment.

_Object Map_ - control information to allow the 'Multics machine' (e.g., the linker) and the Multics standard service system tools to recognize the structure of the object segment.

The generated information items listed above are not stored, intermixed, within a monolithic object segment. Rather, the object segment is structured into four _sections_, named _text_, _definition_, _linkage_ and _symbol_. A section is an array of words.

The object segment is a concatenation of these four sections, in the following sequence,

text || definition || linkage || symbol

the length of all but the last (i.e., symbol) section must be an even number of words.

The assignment of any item of generated code to one of the four sections is decided on the basis of such considerations as access attributes and efficient resource management. The rules of assignment are as follows.

_Text Section_ - contains only the pure (non selfmodifying) executable part of the object program; that is, instructions and read-only constants. It may also contain relative pointers into the definition, linkage and symbol sections as described below.

_Definition Section_ - contains only non-executable read-only symbolic information which is intented for the purpose 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 Multics Binder) be structured into _definition blocks_, which are threaded together.

_Linkage Section_ - contains the impure (i.e., modified during the program's execution) parts of the program 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 variables.
b) internal storage of the type called "own" in ALGOL and "internal static" in PL/1.

_Symbol Section_ - named so 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 three 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 (non-writable) information only.

During the execution of an object program, the Text, Definition and Symbol sections are sharable among several processes; the Linkage section is copied into each process' memory space so that each copy is a per-process data base, to be discarded upon process termination. The original linkage section serves as a copying template only.

## 2. DATA STRUCTURES

This section describes the main data formats and structures which may be encountered within the four sections of an object segment. Definitions are given in PL/1. Most structures defined below have a 'decl_vers' item, a constant which designates the format of the structure; when the structure is modified, so is the constant, allowing system tools to differentiate between concurrent incompatible versions of a single structure. Normally, whenever the structure is modified, the 'decl_vers' item is incremented by one. All structures as defined in this document have the declaration version number set to the constant '1', unless otherwise specified.

### 2.1. The Text Section

The text section is basically unstructured, containing the machine language representation of some symbolic language algorithm as compiled by the appropriate language processor.

Three (optional) items, however, which may appear within the text section have standard formats. They are a) the entry sequence, b) the 645 follow-on gate procedure entry vector, and c) the argument list used in inter-procedure calls.

#### 2.1.1. The Entry Sequence (NEW)

The entry sequence is mandatory for executable object segments (as opposed to compiled data bases in object segment format); there must be an entry sequence for every procedure entry point. The entry sequence has the following format.

```
declare  1 entry_sequence aligned,
         2 symbol_ptr bit(18) unaligned,
         2 descriptor bit(1) unaligned,
         2 unused bit(17) unaligned,
         2 save_sequence(n) bit(36) aligned;
```

symbol_ptr - pointer (relative to the base of the definition section) to the symbolic definition 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.

descriptor - set to "1"b if the entry point's formal definition (see section 2.2.2) contains descriptors for the arguments of this entry point; this information is redundant, duplicated in the formal definition.

save_sequence - is an array of *n* words containing the standard save sequence code.

*note*: the value (i.e., offset within the text section) of the entrypoint corresponds to the address of the 'save_sequence' item. If 'ent_offset' is the value of entrypoint "start", then the symbol_ptr pointing to definition "start" is located at (ent_offset-1).

## 2.1.2. The Gate Procedure Entry Vector (NEW)

A 645 follow-on gate procedure can only be entered at the first *n* locations at the bottom of the segment (i.e., offsets $(0)^8$ through $(n-1)^8$), where each location corresponds to a numbered entrypoint. The validation of the value *n* is performed by the hardware.

*Note*: The following formats relate to structures whose values are computed at run time and allocated in the procedure's current stack frame. They are nevertheless described within this section because they are logically indivisibly related to the text section and constitute part of the procedure's current activation record.

## 2.1.3. The Argument List

The argument list is a structure, (normally) allocated in the procedure's current stack frame, whose value may be computed at run time. It may have the following format.

```
declare  1 arglist aligned,
           2 n_args bit(18) unaligned,
           2 code bit(18) unaligned,
           2 n_descr bit(18) unaligned,
           2 unused bit(18) unaligned,
           2 arg_ptr(n) pointer,
           2 parent_stack_ptr pointer, (OPTIONAL)
           2 descr_ptr(n) pointer; (OPTIONAL)
```

n_args - a (fixed bin(17)) positive integer whose value is (2**n*) where *n* is the number of arguments in this list.

code - a (fixed bin(17)) positive integer whose value has the dual purpose of a) differentiating between an older (EPL) format and the present (PL/1) format of the argument list, and b) defining whether or not the optional 'parent_stack_frame' item is allocated within the argument list. The code can assume one of the following values.

4 -> this is a PL/1 type argument list.
8 -> this is a PL/1 type argument list featuring an optional 'parent_stack_ptr'.

n_descr - a (fixed bin(17)) positive integer whose value is either zero, indicating that this list contains no optional argument descriptors, or (2*n) indicating that there are n descriptors corresponding to the n arguments; descriptor i corresponds to argument i.

arg_ptr - a pointer (ITS pair) to the base of a variable.

parent_stack_ptr - optional pointer to the stack frame of a nested procedure's immediate parent.

descr_ptr - optional pointer to an argument descriptor.

### 2.1.5. The Stack Frame (NEW)

TO BE SUPPLIED

## 2.2. The Definition Section

For historical reasons, character strings are represented within the Definition Section in ALM 'acc' format which may be defined by the following (free style) PL/1 declaration,

```
declare  1 acc,
         2 char_count bit(9) unaligned,
         2 string char(char_count) unaligned;
```

For the purpose of this document, we shall use the notation 'char acc' to represent an 'acc' string. Note that a) the length of such a varying string is incorporated within the string, and b) 'acc' strings are padded to the right (when necessary) with null characters (000)8.

The definition section contains a number of data structures which are,

### 2.2.1. 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. The definition list is a threaded list of formal definitions, defining those variables within the object program which are made known externally. The list consists of one or more definition blocks, each of which consists of one or more type-3 definitions, and zero or more non type-3 definitions (see below).

```
declare 1 def_header based(p) aligned,
        2 def_list bit(18) unaligned, (NEW)
        2 unused bit(54) unaligned; (NEW)
```

def_list - relative pointer to first definition in definition list.

### 2.2.2. The Definition

The format of a definition is as follows,

```
declare 1 definition based(p) aligned,
        2 forward_thread bit(18) unaligned,
        2 backward_thread bit(18) unaligned, (NEW)
        2 value bit(18) unaligned,
        2 flags, (NEW)
          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, (NEW)
          2 symbol_ptr bit(18) unaligned, (NEW)
          2 segname_ptr bit(18) unaligned, (NEW)
          2 n_args bit(18) unaligned, (NEW)
          2 descriptor(n_args) bit(18) unaligned; (NEW)
```

forward_thread - thread (relative to the base of the definition section) to the next definition. The thread terminates when it points to a 645 word which is all zero. This thread provides a single sequential list of all the definitions within the definition section.

backward_thread - thread (relative to the base of the definition section) to the preceding definition. The thread terminates when it points to a 645 word which is all zero. This thread provides a single sequential list of all the definitions within the definition section.

value - this is the offset, within the section designated by the class variable, of this symbolic definition.

flags - 15 binary indicators to provide additional information about this definition.

- new_format -> "1"b definition has the format described in this document, as distinct from the older definition format.
- ignore -> "1"b this definition does <u>not</u> represent an external symbol and must therefore be ignored by the Multics linker.
- entrypoint -> "1"b this is the definition of an entrypoint (i.e., a variable referenced through a transfer of control instruction).
- retain -> "1"b this definition must not be deleted from the object segment.
- descr_sw -> "1"b this definition includes an array of argument descriptors (i.e., items 'n_args' and 'descriptor(n_args)' contain valid information).

class - this field contains a (fixed bin(3)) code which indicates relative to which section of the object segment the value is, as follows,

- 0 -> text section
- 1 -> linkage section
- 2 -> symbol section
- 3 -> this symbol is a segment name (NEW)

symbol_ptr - pointer (relative to the base of the definition section) to an aligned acc string representing the definition's symbolic name.

segname_ptr - Pointer (relative to the base of the definition section) to the first class-3 (see below) segname definition of this definition block.

note: the following two items may be interpreted only if descr_sw="1"b.

n_args - a positive fixed bin(17) integer whose value corresponds to the number of arguments expected by this external entrypoint.

descriptor - an array of pointers, relative to the base of the text section, which point to the descriptors of the corresponding entrypoint arguments.

In the case of a class-3 definition, which is the segname header of a definition block, the above structure is interpreted as follows,

```
declare  1 segname based(p) aligned, (NEW)
         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_ptr bit(18) unaligned,
         2 defblock_ptr bit(18) unaligned;
```

segname_thread - thread (relative to the base of the definition section) to the next class-3 definition. The thread terminates when it points to a 645 word which is all zero. This thread provides a single sequential list of all class-3 definitions.

defblock_ptr - this thread (relative to the base of the definition section) points to the head of the definition block associated with this segname. Definition blocks (which are each a list of non class-3 definitions threaded together by the forward_thread) are preceded sequentially (within that thread) by zero or more class-3 definitions each of which has its defblock_ptr pointing to the block's first non class-3 definition.

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 defblock_ptr point to the same class-3 definition.

Figure-1 illustrates the threading of definition entries.

## 2.3.3. The Expression Word

The expression word is the item pointed to by the expression pointer of an unsnapped link (see below), and has the following structure,

```
declare   1 exp_word based(p) aligned,
          2 type_pair_ptr bit(18) unaligned,
          2 expression bit(18) unaligned;
```

type_pair_ptr - pointer (relative to the base of the definition section) to the link's type-pair.

expression - a signed fixed binary(17) value to be added to the value (i.e., offset within a segment) of the resolved link.

## 2.2.4. The Type Pair

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

```
declare   1 type_pair based(p) aligned,
          2 type bit(18) unaligned,
          2 trap_ptr bit(18) unaligned,
          2 segname_ptr bit(18) unaligned,
          2 entryname_ptr bit(18) unaligned;
```

type - this is a fixed binary(17) positive integer which may assume one of the following values,

1 -> this is a selfreferencing 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 (NEW), used to define a now obsolete ITB-type link.

3 -> this is a link referencing a specified segment, but no symbolic entryname of the form

segname|0+expression,modifier

4 -> this is a link referencing both a symbolic segmentname and a symbolic entryname, of the form

segname$entryname+expression,modifier

5 -> this is a selfreferencing link having a symbolic entryname, of the form

myself$entryname+expression,modifier

trap_ptr - if non-zero then this is a pointer (relative to the base of the definition section) to a trap-pair.

segname_ptr - is a pointer to the referenced segment; its value may be interpreted in one of two ways, depending on the value of the type item,

a) for types 1 and 5, this item is a fixed binary(17) positive integer code which may assume one of the following values, designating the sections of the selfreferencing object segment,

0 -> selfreference to the object's text section; such a reference is represented symbolically as '*text'
1 -> selfreference to the object's linkage section; such reference is represented symbolically as '*link'
2 -> selfreference to the object's symbol section; such reference is represented symbolically as '*symbol'

b) for types 3 and 4, this item is a pointer (relative to the base of the definition section) to an aligned 'acc' string representation of the referenced segment's symbolic name.

entryname_ptr - is a pointer to the referenced item (i.e., offset within the referenced segment); its value may be interpreted in one of two ways, depending on the value of the type item,

a) for types 1 and 3, this value is ignored and an offset of 0 (zero) is assumed,

b) for types 4 and 5, this item is a pointer (relative to the base of the definition section) to an aligned 'acc' string representation of an external symbol,

## 2.2.5. The Trap Pair

The trap pair is a structure specifying two external symbols (i.e., pointing to two links), the first of which is the call pointer and the second being the argument pointer. During the process of dynamic linking, the Linker -while processing a type pair- may encounter a non-zero trap_ptr; in that case, prior to the snapping of the actual link, the linker first invokes the trap procedure using the specified call and argument pointers. The trap pair is structured as follows,

```
declare  1 trap_pair based(p) aligned,
           2 call_ptr bit(18) unaligned,
           2 argument_ptr bit(18) unaligned;
```

call_ptr - a pointer (relative to the base of the linkage section) to a link specifying the entrypoint of a trap procedure.

argument_ptr - a pointer (relative to the base of the linkage section) to a link specifying the base of an argument list to be passed to the trap procedure.

note: The invokation of a trap procedure involves some technical difficulty in that a standard argument list cannot be used in conjunction with such a call. More specifically, the argument list is an array of pointers, normally residing in the procedure's stack frame, whose value is computed at run time. Argument lists for trap procedures must be provided at compile time (because, by definition, they will be used prior to the first invokation of the trapped procedures) and may therefore contain no pointers, whose values are undetermined at that time. Two possible solutions are recommended.

a) the argument list is prepared at run time by an initialization procedure, and put into an external static data base pointed to by the 'argument_ptr' link.

b) the trap procedure uses a non-standard argument list containing constant values rather than pointers to variables (e.g., the 'datmk_' procedure).

In any case, it is currently impossible for users to specify traps before link using high level languages (e.g., PL/1 Fortran etc.), so that the use of this facility is practically restricted to system programmers using the Multics assembly language ALM.

## 2.3 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 allocated within the linkage section in the following sequence,

header || internal storage || links || traps

with the further restriction that the link structure must begin at an even location (offset) within the linkage section.

### 2.3.1. The Linkage Section Header

The header of the linkage section has the following format,

```
declare  1 linkage_header based(p) aligned,
         2 object_seg fixed binary,
         2 def_section bit(18) unaligned,
         2 first_reference bit(18) unaligned, (NEW)
         2 section_thread pointer,
         2 linkage_ptr pointer, (NEW)
         2 begin_links bit(18) unaligned,
         2 section_length bit(18) unaligned,
          2 object_seg bit(18) unaligned,
          2 combined_length bit(18) unaligned;
```

object_seg - reset to zero.

def_section - a pointer (relative to the base of the object segment) to the base of the definition section.

first_reference - 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. Important: as explained in following note, this item is overwritten in the copied linkage section with an ITS pointer to the object's definition section. Consequently, its value may only be validly interrogated within the original linkage section template.

Note: when the object segment is loaded into memory for the purpose of execution, the impure linkage section is copied into a per-process writable data base (known as the combined linkage section) and the preceding items (which are intentionally allocated to occupy a contiguous pair of words) are overwritten with a pointer variable (645 ITS pair) pointing to the base of the definition section).

section_thread - under certain applications, linkage sections may be threaded together, to form a linkage list; such applications are not discussed within this document. The forward_thread is an absolute pointer to the next linkage section in the list, allowing a list to spread over more than a single segment.

linkage_ptr - is a pointer, set by the linker during the process of copying into the combined linkage section, to the original linkage section within the object segment. It is used by the link unsnapping mechanism.

begin_links - this is a pointer (relative to the base of the linkage section) to the first link (the base of the link structure). The length of the linkage header is known to be set to the fixed value 8, providing an implicit relative pointer to the base of the internal storage area.

section_length - this is a fixed binary(18) positive integer value representing the length, in words, of the linkage section.

object_seg - when the linkage section is copied into the combined linkage section, the segment number of the object segment is put into this item.

combined length - when several linkage sections are combined into a list, this item (of the first linkage section in the list) contains the length of the entire list.

### 2.3.2 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 to it.

### 2.3.3 The Links

This is an array of links, each defining an external symbol referenced by this object segment whose effective address is unknown at compile time and can be resolved only at the moment of execution.

A link must reside on an even address 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 based(p) aligned,
         2 header_pointer bit(18) unaligned,
         2 ignore1 bit(12) unaligned,
         2 tag bit(6) unaligned,
         2 expression_ptr bit(18) unaligned,
         2 ignore2 bit(12) unaligned,
         2 modifier bit(6) unaligned;
```

header_pointer - is a backpointer (relative to the head of the linkage section) 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.

ignore1 - unused. Reset to zero.

tag - a constant (46)8 which represents a 645 fault tag 2 and distinctly identifies an unsnapped link. The snapped link (ITS pair) has a distinct (43)8 tag.

expression_ptr - pointer (relative to the base of the definition section) to the expression structure defining this link.

ignore2 - unused. Reset to zero.

modifier - a 645 address modifier.

Figure-2 illustrates the structure of a link.

### 2.3.4. The First-Reference Traps

It is sometimes desired to effect some special initialization of an object segment when it is first referenced for execution (i.e., linked to) in a given process, for example in order to complement the object segment with process dependent information, such as a segment number. The array of first-reference traps contains relative pointers to links defining procedures to be invoked upon first reference within a process, and corresponding links to specify argument pointers for such invokations (if any). Normally, a procedure may have a single initialization trap, however bound segments may specify several. If item 'first_reference' in the linkage section header is "0"b then no such initialization is required; a non-zero value of that item is a relative pointer to the array of traps, and indicates that initialization is required.

```
declare  1 fr_traps based(p) aligned, (NEW)
         2 decl_vers fixed bin,
         2 n_traps fixed bin,
         2 array(n_traps) aligned,
         2 call_ptr bit(18) unaligned,
         2 arg_ptr bit(18) unaligned;
```

decl_vers - 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 incompatible versions of a single structure.

n_traps - specifies the number of trap pointers in this structure. An object segment, such as a bound object, may have several initialization traps to be invoked.

call_ptr - a pointer (relative to the base of the linkage section) to a link specifying an initialization procedure to be invoked by the linker upon first reference to this object within a given process.

arg_ptr - if unequal "0"b, this is a pointer (relative to base of linkage section) to a link specifying an argument list for matching 'call_ptr'.

## 2.4. The Symbol Section

The symbol section consists of one or more symbol blocks, followed by an object map, which are allocated contiguously and threaded (beginning with the object map) to form a single list. It terminates with a single 645 word containing a (left adjusted) 18-bit pointer (relative to the base of the object segment) pointing to the object map. This pointer must always constitute the last word of an object segment. The size (in words) of the object segment is a quantity which may be obtained from the Multics file system. Using this value, it is possible to locate the segment's object map (through this pointer) which in its turn contains all the information necessary in order to identify and access the divers components of the object segment. Knowledge of the object map is the key to the decoding of an object segment, and the convention by which the last word points to the object map provides that key.

The symbol section contains a significant number of variable length character strings which should be directly accessible, but which (for the sake of economy) should preferably be stored in packed format. In order to achieve such storage organization, strings within the symbol section may be pointed to by a string pointer, which contains both offset and length of the string in packed form,

```
          declare  1 stringpointer aligned, (NEW)
                   2 offset bit(18) unaligned,
                   2 length bit(18) unaligned;
```

where offset is a pointer (relative to the base of the symbol block) to the first character of the aligned string, and length is a (fixed binary(17)) positive integer representing the length of the string in characters. This representation allows easy access to the string by using the PL/1 built in functions 'addrel', 'fixed' and 'substr'. In the following description, we shall use the notation 'stringpointer' to denote such a pointer; a stringpointer is null if its value is all zero.

### 2.4.1 The Object Map (NEW)

The object map is a fixed length structure residing at the very end of the object segment. It contains all the necessary structural information pertaining to the object segment.

```
declare   1 object_map based(p),
          2 decl_vers fixed bin,
          2 identifier char(8) aligned,
          2 text_offset bit(18) unaligned,
          2 text_length bit(18) unaligned,
          2 definition_offset bit(18) unaligned,
          2 definition_length bit(18) unaligned,
          2 linkage_offset bit(18) unaligned,
          2 linkage_length bit(18) unaligned,
          2 symbol_offset bit(18) unaligned,
          2 symbol_length bit(18) unaligned,
          2 first_block bit(18) unaligned,
          2 number_of_blocks bit(18) unaligned,
          2 format aligned,
            3 bound bit(1) unaligned,
            3 relocatable bit(1) unaligned,
            3 procedure bit(1) unaligned,
            3 unused bit(15) unaligned,
          2 map_ptr bit(18) aligned;
```

decl_vers - 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 incompatible versions of a single structure.

identifier - must be the constant "obj_map".

text_offset - offset (relative to the base of the object segment) of the text section.

text_length - a fixed binarry(17) positive integer representing the length in words of the text section.

definition_offset - analogous to text_offset

definition_length - analogous to text_length

linkage_offset - analogous to text_offset

linkage_length - analogous to text_length

symbol_offset - analogous to text_offset

symbol_length - analogous to text_length

first_block - Pointer (relative to the base of the symbol section) to the most recent symbol block. An object segment may have one or more symbol blocks which are threaded on a list in reverse chronological order (i.e., newest block is first on the list).

number_of_blocks - this is a (fixed binary(17)) positive integer displaying the number of symbol blocks within this symbol section.

bound - this is a bound segment

relocatable = "1"b -> this object segment has relocation information in its first symbol block.

procedure - "1"b -> this is an executable object program;
"0"b -> this is a data base.

map_ptr - this is a pointer, relative to the base of the object segment, to the object map; as mentioned before, this item must reside in the last word of the object segment.

### 2.4.2. The Symbol Block Header (NEW)

The symbol block has two main functions, a) to document the circumstances under which the object was created, and b) serve as a repository for information which does not belong in any of the other three sections (e.g., 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 also contain more than a single symbol block, for example in the case of a bound object, where in addition to the symbol block describing the object's creation by the binder, there is also a symbol block for each of the component objects. The symbol section is designed so that symbol blocks may be dynamically appended to, or deleted from it, such as in the case of the debugger which allocates itself a symbol block in order to store in it breakpoint information. The size and structure of a symbol block are variable, depending upon their purpose. All symbol blocks have a standard fixed format header, as follows.

```
declare  1 symbol_block_header based(p) aligned,
         2 decl_vers fixed bin,
         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 stringpointer,
         2 userid stringpointer,
         2 comment stringpointer,
         2 text_boundary bit(18) unaligned,
         2 stat_boundary bit(18) unaligned,
         2 source_map bit(18) unaligned,
         2 area_pointer bit(18) unaligned,
         2 sectionbase_backpointer bit(18) unaligned,
         2 block_size bit(18) unaligned,
         2 next_block_thread bit(18) unaligned,
         2 rel_text bit(18) unaligned,
         2 rel_def bit(18) unaligned,
         2 rel_link bit(18) unaligned,
         2 rel_symbol bit(18) unaligned,
         2 default_truncate bit(18) unaligned,
         2 optional_truncate bit(18) unaligned;
```

decl_vers - 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 incompatible versions of a single structure.

identifier - symbolic code to define the purpose of this symbol block. It may assume one of the following values.

```
    "symbtree" -> compiler symbol tree
    "bind_map" -> bind map
    "dbbreak" -> debug breakpoint information
```

gen_version_number - a positive integer designating the version of the generator which was used in compiling this object program. The policy regarding this version number is that whenever a generator is substantially modified, such as the addition of new capabilities or the generation of new object code patterns, this number has to be incremented by one. It is used mainly by system tools which sometimes have to be cognizant of the code generation peculiarities of a given compiler.

gen_creation_time - a calendar clock reading specifying the date/time at which this generator was created.

object_creation_time - a calendar clock reading specifying the date/time at which this symbol block was generated.

generator - symbolic code defining the processor which generated this symbol block. It may assume one of the values in the following list (which is subject to change or expansion).

"alm"
"pl1"
"fortran"
"binder"
"debug"

gen_version_name - the generator's version in directly printable character string form, such as,

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

this string is displayed by various system tools. 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 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 that version. It is mandatory that the generator name be updated whenever the generator is installed for public use.

userid - the standard Multics identifier of the user in behalf of whom this symbol block was created.

comment - it is sometimes desirable to put certain factual information concerning the generator (e.g., certain code generation peculiarities) of perhaps the actual process of object program generation (e.g., warning about non fatal errors encountered during compilation, or warning concerning certain defaults applied by the generator) into the object segment. The comment is displayed by certain system tools, and may be of special interest, for example, when a decision has to be made concerning the suitability of a given object segment for official installation in the system libraries.

text_boundary - for specialized programs, it is sometimes necessary that the text section begin on a predetermined boundary (e.g., 0 mod 64 address); this is an integer which defines this boundary. Its default value is 2 (0 mod 2 address).

stat_boundary - same as text_boundary, for internal static. Its default value is 2.

source_map - a pointer (relative to the base of the symbol block) to a source_map structure (see 2.4.3) defining the pathnames of the source files. If no source map is provided, this pointer is reset to "0"b.

area_pointer - Pointer (relative to the base of the symbol block) to the actual symbol block information (e.g., symbol tree, bind map etc.).

sectionbase_backpointer - pointer (relative to base of symbol block) to base of symbol section. This is a negative quantity.

block_size - a (fixed binary(17)) integer value representing the size of the symbol block (including header) in words.

next_block_thread - thread (relative to base of symbol section) to next symbol block.

rel_text - pointer (relative to base of symbol block) to text section relocation information, as defined below.

rel_def - pointer (relative to the base of the symbol block) to definition section relocation information.

rel_link - pointer (relative to base of symbol block) to linkage section relocation information.

rel_symbol - pointer (relative to base of symbol block) to symbol section relocation information.

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

optional_truncate - offset (relative to base of symbol block) starting from which the binder may optionally truncate non-essential parts of the symbol tree in order to achieve maximum reduction in size of bound object segment.

## 2.4.3. The Source Map

The source map is a structure defining the source segments used to originate this object segment, as follows.

```
declare   1 source_map aligned based(p),
            2 decl_vers fixed bin,
          2 size fixed bin,
          2 map(size) aligned,
            3 pathname stringpointer,
            3 uid fixed bin,
            3 dtm fixed bin(71);
```

decl_vers - 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 incompatible versions of a single structure.

size - the number of entries in the "map" array (i.e., number of source files defined in this structure).

pathname - a stringpointer specifying the full pathname (treename) of the source segment.

uid - the unique identifier of the source segment's branch at compile time.

dtm - the date-time modified from the source segment's branch.

### 2.4.4. 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 (non relocatable) 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 correspondance between the 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 based(p),
        2 decl_vers fixed bin,
        2 n_bits fixed bin,
        2 relbits bit(n_bits) aligned;
```

decl_vers - 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 incompatible versions of a single structure.

n_bits - the size of the string of relocation bits.

relbits - the packed string of relocation bits.

Following is a tabulation of the possible codes and their corresponding relocation types,

| | | |
|---|---|---|
| "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 (NEW) |
| "11001"b | -> | Internal Storage 15 (NEW) |
| "11010"b | -> | Self Relative |
| "11011"b | -> | Unused |
| "11100"b | -> | Unused |
| "11101"b | -> | Unused |
| "11110"b | -> | Expanded Absolute (NEW) |
| "11111"b | -> | Escape |

Absolute - do not relocate

Text - use text section relocation counter

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.

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).

Negative Link 18 - same as above

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 a 645 instruction featuring a base/offset address field.

Definition - indicated that the halfword contains an address which is relative to the base of the definition section.

Symbol - use symbol section relocation counter.

Negative Symbol - same as above

Internal Storage 18 - use internal storage relocation counter on the entire 18-bit halfword.

internal Storage 15 - use internal storage relocation counter on the low order 15-bits of the halfword.

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 5 bits of relocation code are immediately followed by a fixed length 10-bit field which is a (fixed binary(10)) positive count of the number of contiguous halfwords all having an absolute relocation. Evidently, usage of the expanded absolute code can be economically justified only if the number of contiguous absolute halfwords exceeds 4.

Escape - reserved for possible future use.

Figure-3 illustrates the overall structure of the symbol section.

### 2.4.5. The PL/1 Symbol Block

TO BE SUPPLIED

### 2.4.6. The ALM Symbol Block

TO BE SUPPLIED

### 2.4.7. The Binder's Symbol Block

The binder's symbol block contains the bind map, describing the relocation values assigned to the various sections of the bound component object segments. The block consists of a variable length structure, followed by an area in which variable length symbolic information is stored. The format of the bindmap structure is,

```
declare  1 bindmap based(p) aligned,
         2 decl_vers fixed bin,
         2 n_components fixed bin,
         2 component(n_components) aligned,
           3 name stringpointer,
           3 generator_name char(8) aligned,
           3 text_start bit(18) unaligned,
           3 text_length bit(18) unaligned,
           3 stat_start bit(18) unaligned,
           3 stat_length bit(18) unaligned,
           3 symb_start bit(18) unaligned,
           3 symb_length bit(18) unaligned,
           3 defblock_ptr bit(18) unaligned;
```

decl_vers - 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 incompatible versions of a single structure.

n_components - number of component objects bound within this bound segment.

component - variable length array featuring one entry per bound component object segment.

name - pointer 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). The stringpointer is relative to the base of the bindmap structure.

generator_name - the name of the generator which created this component object segment.

text_start - (fixed binary(17)) integer value of the component's text section relocation counter.

text_length - (fixed binary(17)) integer value of the component's text section's length.

stat_start - relocation counter for component's internal static.

stat_length - length of component's internal static.

symb_start - relocation counter for component's symbol section.

symb_length - length of component's symbol section.

defblock_ptr - 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).

## 2.4.8. Debug's Symbol Block

TO BE SUPPLIED

## 3. GENERATED CODE

This section describes those parts of the generated code (other than the structural parts discussed in section 2) which have to conform to a systemwide standard because they interface with system tools such as the binder, the default error handler, debug etc.

### 3.1 The Text Section

The text section contains a number of sequences where it is advantageous to have all generators produce identical code patterns, such as the call, save and return sequences. For the purpose of this document, however, only the entry sequence and the generated relocation codes are of interest.

#### 3.1.1. The Entry Sequence (NEW)

The entry sequence must fulfil two requirements, a) 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 b) that the save 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.

#### 3.1.2. The Relocation Codes (NEW)

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 a direct (i.e., unindexed) reference to a link.

Link 15 - may only appear within the address field of a (base/offset) type instruction (bit29="1"b). The instruction must not be indexed, and must not contain a "10"b tm modifier. Also, the following instruction codes may not have this relocation code,

STBA $(551)_8$
STBQ $(552)_8$
STCA $(751)_8$
STCQ $(752)_8$

Note: the peculiar restrictions imposed upon the link-15 and int-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 op-code (right hand) portion of the instruction word. Therefore, these relocation codes must only be specified in a context which is comprehensible to the 645 control unit.

Definition - no restriction

Symbol - no restriction

Internal Storage 18 - no restriction

Internal Storage 15 - may only appear within the address field of a (base/offset) type instruction (bit29="1"b). The instruction must not contain a "10"b tm modifier, however it may be indexed. The instruction codes excluded from the Link-15 relocation may also be used.

Self Relative - no restriction

Expanded Absolute - no restriction

## 3.2. The Definition Section

There are no relocation codes associated with the definition section. Item 'rel_def' in the symbol block header has been provided for the sake of completeness and may be used in the future.

### 3.2.1. Implicit Definitions (NEW)

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.

Additionally, objects created by the binder have the implicit definition "bind_map" which points to the base of the symbol block generated by the binder, relative to the base of the symbol section.

## 3.3. The Linkage Section

The linkage section consists of four distinct blocks: the linkage section header, the internal storage, the links and the first reference traps.. The format and value of the linkage section header are as defined in section (2.3.1).

### 3.3.1. The Internal Storage

The internal storage is a repository for items of the internal static storage class. It may contain data items only; even though access to the linkage section is of the 'rew' type, it may not contain any executable code.

Text - no restrictions

### 3.3.2. The Links

The link area may only contain an array of links as defined in section (2.3.3). 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.

### 3.3.3. The Relocation Codes (NEW)

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.

Link 18 - no restriction

Negative Link 18 - no restriction

Definition - no restriction

Internal Storage 18 - no restriction

Expanded Absolute - no restriction

## 3.4. The Symbol Section

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

### 3.4.1. The Relocation Codes (NEW)

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

## 4. FUNCTIONAL INTERFACES

This section briefly describes a number of the object segment's functional interfaces in order to give the reader some idea as to how certain structures and formats, described in sections (2, 3) are used. Also, a list of standard system tools is provided in order to allow a subsystem or compiler writer to acquaint himself with existing facilities on Multics.

### 4.1. Dynamic Linking

One of the basic principles of Multics is that information is always accessed by its symbolic file system name, and that segments are assigned a machine address (i.e., segment number) at the moment of execution only. It follows that any inter-segment reference must be resolved prior to its execution into a machine address which is a priori unknown. Certain computer systems require that such address resolution be performed, prior to execution, by a process commonly known as "loading", which may be thought of as a "post-compilation" in which several independently compiled procedures are assembled into a single procedure in which all previous symbolic inter-procedure references are converted into internal relative addresses.

In Multics, such loading is unnecessary because the dynamic linking mechanism allows symbolic references to be evaluated and resolved whenever they are encountered during execution. A hardware register, known as the linkage pointer (lp) is always set to point to the base of the currently executing procedure's linkage section. All references to external symbols are made in the form (lp|n,*) where n is a relative offset within that procedure's linkage section, and notation ',*' indicates indirection (i.e., address substitution). Location (lp|n) contains an unsnapped link, as defined in section (2.3.3), which features a linkfault (46)8 tag. When the processor attempts to execute the indirection and recognizes the fault tag (46)8, execution is interrupted and the processor faults (i.e., forces control) to the Multics linker.

Note: In the following description of the linking mechanism, reference is made to items defined in sections 2.2.2, 2.2.3, 2.2.4 and 2.3.3. The reader may wish to consult Figure-2 which illustrates the structure of a link.

The linker's only input is a pointer to the unsnapped link which initiated the linkage fault. By using the link's 'header_pointer' the linker is able to calculate the address of the linkage section header which in turn contains in its first two words an ITS pointer to the object segment's definition section (this pointer is set when the procedure is referenced for the first time, as is explained below).

Let us name the pointer to the definition section defp; the address calculation

addrel(defp, expression_ptr)

produces a pointer to the link's expression word. Given a pointer to the expression word, the address calculation

addrel(defp, type_pair_ptr)

produces a pointer to the link's type-pair, whereupon in turn address calculations

addrel(defp, segname_ptr)

and

addrel(defp, entryname_ptr)

yield pointers to the respective 'acc' strings which define the external symbol.

The linker first interrogates the 'trap_ptr' item in the link's type-pair, and if that item's value is unequal to "0"b then the linker effects a call to (lp|(trap_ptr),*), a call which in turn may provoke a linkage fault (in Multics, dynamic linking may be recursive).

If the 'trap_ptr' is null (or upon return from the trap procedure) the linker proceeds to obtain a pointer to the referenced object segment. For link types 1 and 5 (selfreferencing links) this is a pointer to the referencing procedure. For link types 3 and 4 the pointer is obtained by calling the Multics file system with the symbolic 'segmentname' portion of the external symbol. The linker is now in possession of the segment number portion seg# for the referenced symbol.

The linker also obtains from the file system a value length which is the length (in words) of the referenced object segment. By convention, (length-1) is the offset within the object segment of a pointer to the object map, which contains the offset of the referenced object's definition section. The linker computes a pointer to the target definition section, searches it, and locates the definition for 'entrypoint' which designates the offset of that symbol within the object segment. Going back to the link's expression word, the linker performs the computation (offset+expression) to obtain the final relative address portion of the referenced symbol. It now inserts values seg# and offset into the corresponding 'header_pointer' and 'expression_ptr' of the unsnapped link, changes the link's tag to (43)8 and thus converts the original unsnapped link into a valid (executable)

ITS pointer, whereupon the referencing procedure's execution is resumed at the point of interruption.

By converting the original linkfault into an ITS pointer it is assured that only the very first reference to an external symbol will invoke the dynamic linking mechanism, and the associates cost of linking. Future references to (lp|n,*) will be directly executed.

By definition, an executable object segment is pure (non-selfmodifying) procedure and may not be altered. As we have seen, the process of dynamic linking requires that an unsnapped link be overwritten with an ITS pointer; also, that ITS pointer contains a seg# which may assume different values depending upon the circumstances under which linking took place. Therefore, whenever the linker attempts to link to an object segment which has never before been referenced within that Multics process, it initiates that segment (i.e., requests the file system to make the segment known within that Multics process under some seg#) and copies its entire linkage section into a writable database known as the combined linkage section. The (lp) register will always point to the linkage section copy, and it is this copy which is modified during the procedure's execution. The process of copying includes the appropriate setting of the 'definition_ptr' (words 0&1), 'linkage_ptr' and 'object_seg' items in the copied linkage section header.

It is sometimes desirable to reverse the process of dynamic linking (unsnap a link) and restore the original linkfault information. Given an offset n to a link in the combined linkage section, unsnapping is trivially achieved by locating the original linkage section in the object segment through the 'linkage_ptr' item in the copied linkage section header, and by overwriting the snapped link with its original value found at

addrel(linkage_ptr, n)

Figure-4 is a flow chart illustrating the overall logic of the linker.

## 4.2. Binding

Dynamic linking is a very useful and powerful capability; it provides the casual user with the convenience of not having to explicitly assemble all of the modules related to his program and "load" them before being able to execute it. Rather, he needs only to be concerned with specific modules which are of interest to him, leaving it up to Multics to locate and link to all other modules which may be either his own, or perhaps library procedure provided as standard tools. Moreover, he need not even be aware of certain modules which are invoked by the system in his behalf.

Sometimes, however, a large subsystem which by right should be coded as a single procedure is in effect subdivided into distinct smaller modules, mostly for reasons of coding (and debugging) convenience. This collection of procedures may now be executed, and will be interlinked by the dynamic linking mechanism. In this case, however, it is known in advance that this collection of physically distinct procedures effectively forms a single logical unit. The cost of dynamic linking, no matter how trivial it may be, will be incurred whenever this subsystem is invoked for the first time by some Multics process. Suppose that we have a compiler named 'comp$comp' which was coded modularly in n distinct modules, each of which features an average of m entrypoints; further suppose that in order to execute the compiler all entrypoints must be linked to by the n modules. The cost of a single compilation will thus be increased by the overhead cost of invoking n*m linkage faults, whereas the only linkage fault that needs to be taken is that of linking to 'comp$comp', all others being internal to the compiler and unnecessary, in the sense that the compiler's modularity is a convenience to the writer of the compiler but an unnecessary and expensive penalty to the user.

The Multics binder is a "post processor" which, given an input of n object segments combines and reduces them into a single new 'bound' object segment. One of the functions of binding is to reduce all internal intersegment references from linkfaults to relative internal addresses. Thus, by binding all components of our compiler, we would produce a new object segment named 'comp$comp' whose execution provokes none of the previous n*m linkage faults.

Another reason for binding is that in a paged virtual memory such as Multics, n distinct object segment would incur the extra expense of an average 1/2 a page of lost storage per segment. By binding many component objects (even if they perhaps are only marginally related to one another) one may make substantial gains in storage space.

By binding several object segments, whether related or not, one loses none of the capabilities associated with those object segments in their free standing form. The only discernible effect of binding is that the storage requirements of the bound object segment are less then the combined storage requirements of all the component object segments, and that any internal intersegment references will be pre-linked automatically. Functionally, the execution of a collection of bound object segments is guaranteed to be identical to the execution of those same object segment in free-standing form.

## 4.3. Naming Conventions

Multics segments have symbolic names which may be from 1 to 32 characters long. By convention, such names may be compound, consisting of a concatenation of two or more sub-names where the point of concatenation is flagged by the insertion of a "." character; the number of sub-names within a compound name is limited only by the imposed maximum total length of 32 characters.

It is often desirable to give similiar names to two or more logically related segments. For example, if we have a segment containing the symbolic source language of some program 'prog' and we compile it to produce two more segments, namely the object segment and a segment containing a printable listing of the compilation, we would like to indicate that these two new segments are in effect a deriviative of 'prog' and give them names in which the symbol 'prog' is featured.

By convention, it is always the object segment which is given the primary name 'prog'. All other related segments are given compound names consisting of the primary (first sub-) name 'prog' and one or more standard suffixes. Thus if the source language in our example is PL/1, the segment containing that source code is by convention named 'prog.pl1', and the listing segment produced by the PL/1 compiler is named 'prog.list'. By using this systemwide convention, we may now invoke the PL/1 compiler by typing

```
pl1 prog
```

and the compiler will automatically construct the name 'prog.pl1' and locate that segment which it knows by convention to contain the source code for 'prog'.

## 4.4. Standard System Tools

TO BE SUPPLIED