TO: C. Clingen
F. Corbató
R. Fenichel
R. Freiburghouse
J. Gintell
M. Meer
N. Morris
J. Saltzer✓
V. Voydock
S. Webber
B. Wolman

FROM: Mike Spier

DATE: 4/27/71

SUBJECT: The Multics Standard Object Segment

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.

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

1.3 Naming Conventions

1. OVERVIEW

The Multics standard object program is the executable hardware-level (i.e., machine code) representation of some higher language algorithm, produced by the appropriate language processor. Physically, it is a single fixed-length array of words, which always resides at the base of a distinct segment, and is commonly known as 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 interpretative 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.

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 a fixed length array of words. The object segment is a concatenation of these four section, 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 affectation 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 affectation 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, which consist of two types of data a) linkpairs (described further on) which are modified at run time by the Multics linker to contain the machine address of external variables, and 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.

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.

2.1. The Text Section

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.~~ zeros.

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

2.2.1. 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,
         2 value bit(18) unaligned,
         2 flags,
           3 new_format bit(1) unaligned,
           3 entrypoint bit(1) unaligned,
           3 retain bit(1) unaligned,
           3 ignore bit(1) unaligned,
           3 unused bit(5) unaligned,
         2 class bit(9) unaligned,
         2 symbol_ptr bit(18) unaligned,
         2 segname_ptr bit(18) unaligned,
         2 n_args bit(18) unaligned,
         2 descriptor(n_args) bit(18) unaligned;
```

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 sequemtial 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 - 9 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.
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.
ignore -> "1"b this definition does <u>not</u> represent an external symbol and must therefore be ignored by the Multics linker.

class - this field contains a 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

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.

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,
         2 forward_thread bit(18) unaligned,
         2 backward_thread bit(18) unaligned,
         2 segname_thread bit(18) unaligned,
         2 flags bit(9) unaligned,
         2 class bit(9) 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 (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 definition.

Figure-1 illustrates the threading of definition entries.

2.2.2. 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.

2.2.3. 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

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 ~~5~~ 3 this value is ignored and an offset of 0 (zero) is assumed.

b) for types ~~3~~ 4 and ~~4~~ 5 this item is a pointer (relative to the base of the definition section) to an aligned 'acc' string representation of an external symbol.

2.2.4. 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.

2.3 The Linkage Section

The Linkage section is substructured into three 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 and c) a variable length structure of links. These three components are allocated within the linkage section in the following sequence,

linkage header || internal storage || links

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 ignore1 bit(18) unaligned,
         2 forward_thread pointer,
         2 backward_thread pointer,
         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.

ignore1 - unused

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).

forward_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.

backward_thread - is an absolute pointer to the preceding linkage section in the list.

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 was 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

TO BE SUPPLIED

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.

2.4. The Symbol Section

The symbol section consists of a section header, followed by one or more symbol blocks, allocated contiguously and threaded to form a single list, and terminated with a single 645 word containing an 18-bit pointer (relative to the base of the object segment) to the base of the symbol section. 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 object segment's symbol section 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 symbol table is the key to the decoding of an object segment, and the convention by which the last word points to the symbol section provides that key.

The symbol section contains a significant number of variable length character strings which should preferably be directly accessible, but which (for the sake of economy) should 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 string_pointer aligned,
         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 Symbol Section Header

The symbol section header is a fixed length structure residing at the base of the symbol section. It contains all the necessary structural information pertaining to the object segment.

```
declare  1 symbol_section_header based(p),
         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 procedure bit(1) unaligned,
          3 gate bit(1) unaligned,
          3 execute_only bit(1) unaligned,
          3 mastermode bit(1) unaligned,
          3 relocatable bit(1) unaligned,
          3 unused bit(13) unaligned,
          3 call_delimitor bit(18) unaligned,
        2 objectname char(32) aligned;
```

identifier - must be the constant "symbsect".

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 historical order.

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

procedure = 0 -> this is a non-executable object segment (e.g., an ALM database).
1 -> this is an executable procedure.

gate = 1 -> this object segment is generated in gate format.

execute_only = 1 -> this object segment is generated in execute only format.

mastermode = 1 -> this object segment is generated in mastermode format.

relocatable = 1 -> this object segment has relocation information in its single symbol block

call_delimitor = if the gate flag is "1"b then this item is a fixed binary(17) value representing the gate's call delimitor.

objectname - the segmentname of this object

2.4.2. The Symbol Block Header

The symbol block has two main functions, a) to document the circumstances under which the object was created, and b) allow for modular on-line expansion of the object segment (e.g., by debug). 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 one 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 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),
          2 identifier char(8) aligned,
          2 generator char(8) aligned,
          2 object_creation_time fixed bin(71),
          2 gen_creation_time fixed bin(71),
          2 gen_version_number fixed bin,
          2 gen_version_name stringpointer,
          2 userid stringpointer,
          2 source_map bit(18) unaligned,
          2 n_source_files bit(18) unaligned,
          2 block_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_link bit(18) unaligned,
          2 rel_symbol bit(18) unaligned,
          2 mini_truncate bit(18) unaligned,
          2 maxi_truncate bit(18) unaligned;
```

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

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

"alm"
"pl1"
"fortran"
"apl"
"bcpl"
"binder"
"debug"

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

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

gen_version_number - a positive integer defining the generator's version number.

gen_version_name - the generator's version in directly printable character string form.

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

source_map - a pointer (relative to the base of the symbol block) of an array of stringpointers defining the pathnames of the source files.

n_source_files - size of the source_map array.

block_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 - fixed binary(17) integer representing size of symbol block in words (incl. size of header).

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_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.

mini_truncate - offset (relative to base of symbol block) starting from which control information (such as relocation bits) may be truncated from symbol section, while still maintaining such information as the symbol tree.

maxi_truncate - offset (relative to base of symbol block) starting from which the symbol block may be truncated to achieve maximum reduction in size.

2.4.3. The Relocation Blocks

The relocation information describes 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 one. 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 a 5 or 15 bit item.

There are three distinct blocks of relocation information, one for each of the three object segment sections: text, linkage and symbol; these relocation blocks are known as rel_text, rel_link and rel_symbol, correspondingly. The definition section is not relocatable and consequently has no corresponding relocation block. These are not true symbol section blocks in the sense that they reside within the symbol block of the generator which produced the object segment. Moreover, the relocation blocks must reside at the very end of the generator block so that they may be stripped off when they are no longer needed.

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 n_bits fixed bin(17),
          2 relbits bit(n_bits) aligned;
```

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

```
0     -> Absolute
10000 -> Text
10001 -> Negative Text
10010 -> Link 18
10011 -> Negative Link 18
10100 -> Link 15
10101 -> Definition
10110 -> Symbol
10111 -> Negative Symbol
11000 -> Internal Storage 18
11001 -> Internal Storage 15
11010 -> Self Relative
11011 -> Unused
```

11100 -> Unused
11101 -> Unused
11110 -> Expanded Absolute
11111 -> 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 has 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. Definition sections are not relocatable.

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 count of the number of contiguous halfwords all having an absolute relocation. Evidently, the expanded absolute code is used economically only if the number of absolute halfwords exceeds 4.

Escape - reserved for possible future use.

2.4.4. The PL/1 Symbol Block

TO BE SUPPLIED

2.4.5. The ALM Symbol Block

TO BE SUPPLIED

2.4.6. 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 n_components fixed binary(17),
         2 component(n_components) aligned,
           3 name stringpointer,
           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;
```

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 bndfile).

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).

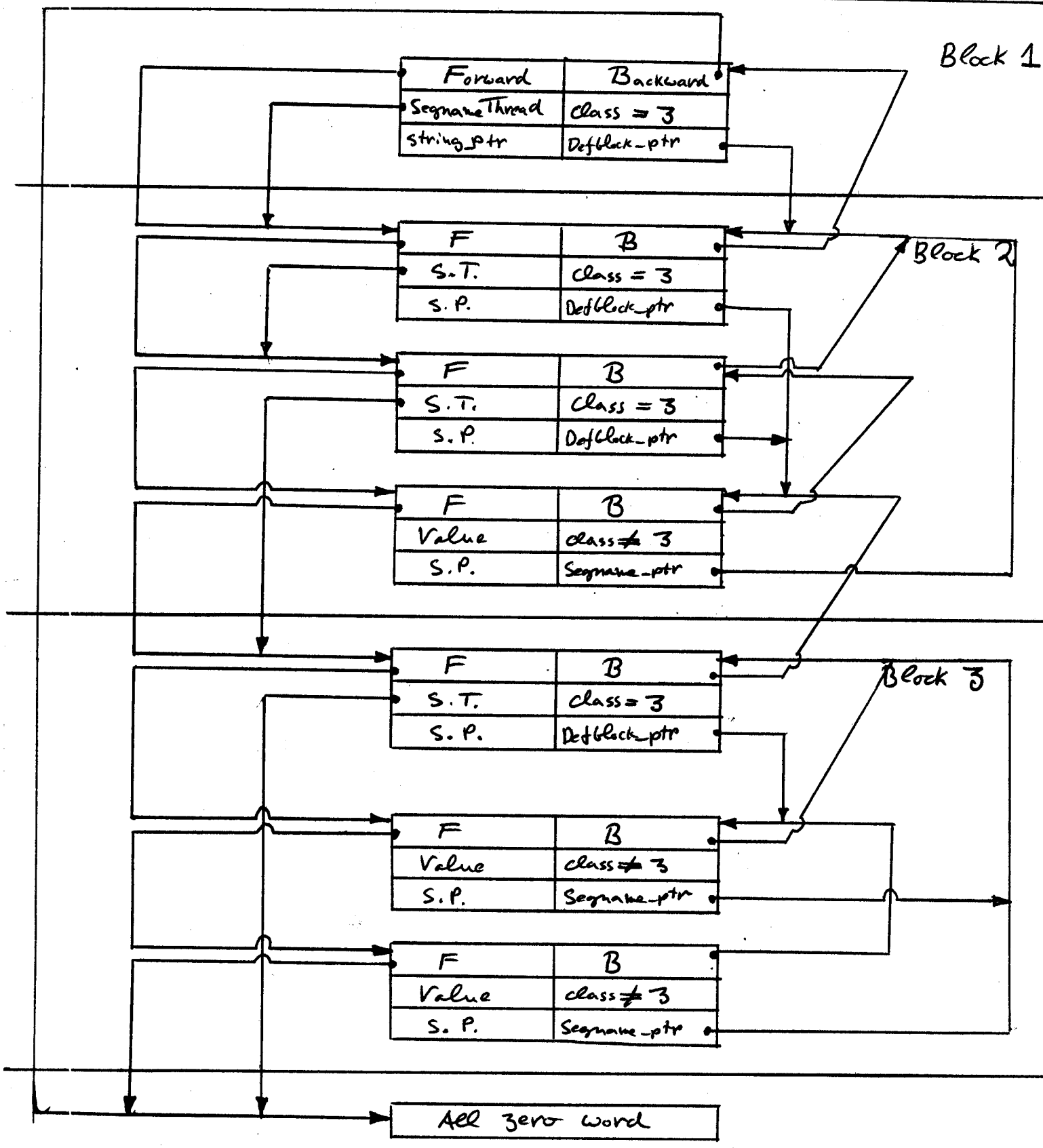

Figure 1: The Definition Section Threads

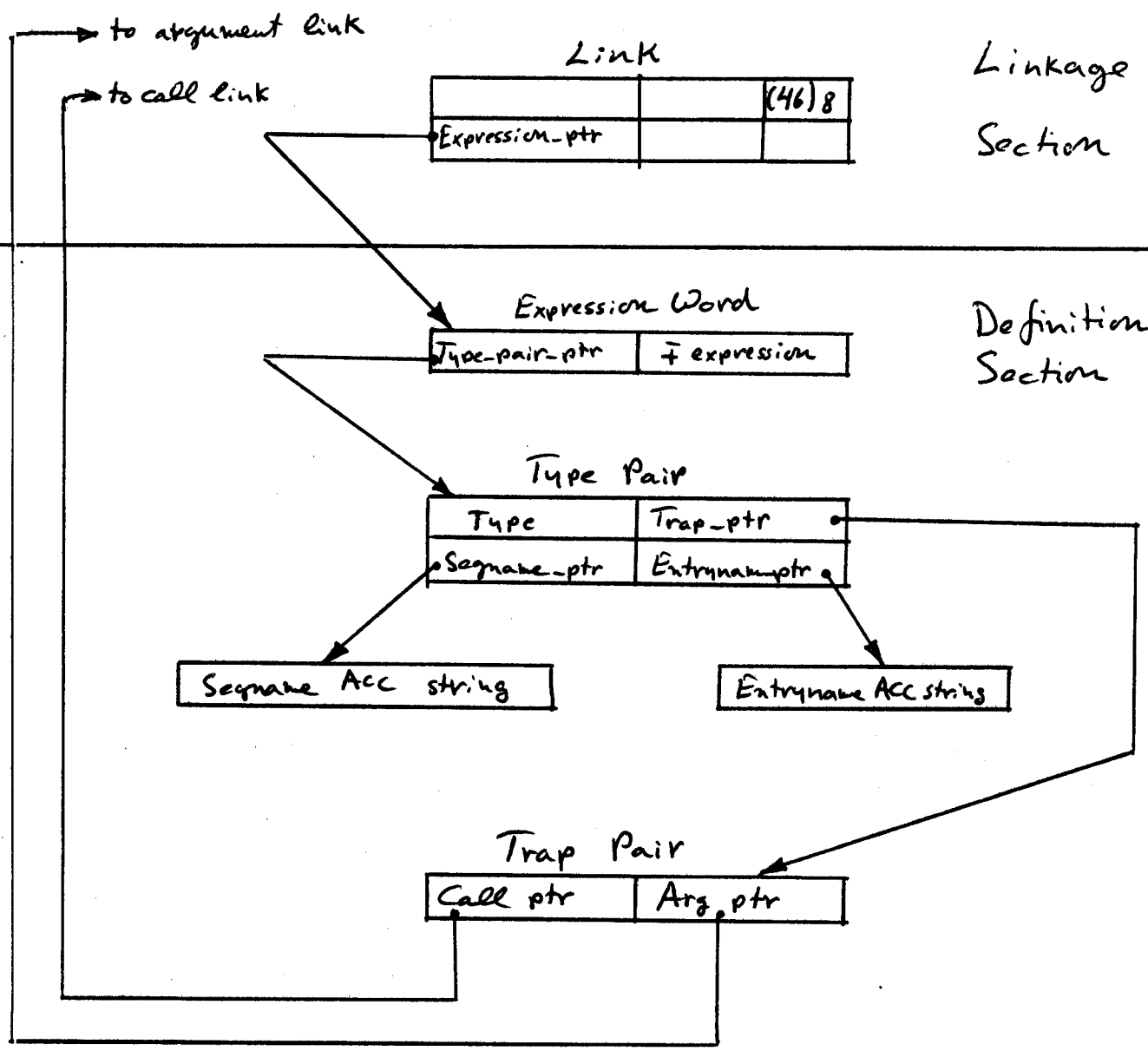

Figure 2: The Structure of a link