MPE OBJECT CODE FORMATS
AN INTRODUCTION TO USL AND PROGRAM FILES

Matthew J. Balander
The B & B Computer Company

ABSTRACT: The format and management of USL and PROGRAM files under the MPE III operating system, running on Hewlett Packard 3000/II and 3000/III computer systems, are presented in this paper. USL files are used to store relocatable binary modules, and PROGRAM files to store fully prepared programs. The presentation is aimed at programmers implementing compilers on HP3000 systems. The reader is assumed to be familiar with the architecture of these systems, and to understand basic concepts of relocatable code, link editing, and so on. The scope of the presentation is intended to provide the reader an adequate background with which to successfully pursue a compiler-writing project.

CONTENTS

1. Introduction

1.1. Overview

All operating systems adopt conventions concerning the formats of object code files. These files must be in correct formats to be processed by system segmenters, linkage editors, and loaders. The MPE operating system defines four types, or formats, of object code files, as follows: user subprogram libraries (USL's), relocatable libraries (RL's), segmented libraries (SL's), and program files (PROGRAM's). Of these, RL's and SL's have rather specialized uses, and their formats are of little interest to the compiler writer. The formats of USL and PROGRAM files, on the other hand, are of great interest. If a compiler is to produce absolute code, it will generate PROGRAM files. If it is to produce relocatable code, it will generate USL files.

This paper presents an overview of the formats of USL and PROGRAM file formats used by MPE III, on HP3000/II and HP3000/III computer systems. It is neither exhaustive nor scrupulously detailed. Readers with some experience in object code formats will not find it difficult to fill in details not included here by examining USL and PROGRAM files.

A word of caution to the reader is appropriate at this point. Hewlett Packard is unco-operative, and seems quite indifferent to the needs of its users to understand MPE conventions. Because of this, all the information in this paper had to be deduced from examination of USL and PROGRAM files. In consequence, although the author believes the information included here to be fully accurate as of the date of this writing, the reader should keep in mind that it may nonetheless include some errors. For the most part, though, it may be used with confidence. The author has written a compiler which generates USL files based on section three of this paper, and a PROGRAM file decompiler based on section two. Both are operating satisfactorily.

1.2. Conventions

A number of conventions are adopted to enable concise explanations and illustrations. These conventions are applied consistently, but occassional, well-marked deviations do occur. The conventions are as follows:

- SPL/3000 notation is used in all cases where illustrative code is provided.

- "P" represents a variable of type integer pointer. In all tables, illustrations, and examples it is assumed to point to the first word of the entity under discussion.

- In the case of single bit fields, "1" is the "on" state, and "0" is the "off" state. Similarly, the on state is the true state, and the off state is the false state.

- Names in PROGRAM and USL files, such as procedure or segment names, are a variable number of words in length. The first byte of a name is always in the P.(8:8) field, the length of the name in bytes in P.(4:4), and various context-dependent information in P.(0:4). The name continues in as many consecutive bytes as needed, beginning with P.(8:8). A name field is always an integral number of words in length. The last byte is thus wasted if the name is an even number of bytes long. In illustrations, the P.(0:8) field will be diagrammed explicitly, but the remainder of the name will be shown simply as a large undivided area. The reader should keep in mind that this represents a variable length field. In the text of the paper, this entire group of fields is referred to as a "name field," and the various parts are not explicitly mentioned unless necessary for the discussion.

- In illustrations, when a word is divided, unless indicated otherwise, it is divided into bytes, or into nibbles. Because of this, no explicit bit field indications are given in illustrations for byte or nibble aligned fields.

2. PROGRAM files

The loader in MPE processes PROGRAM files. In a program file, all relocatable references have been resolved, global storage laid out and initialized, and segmentation completed. The only link editing remaining to be done is to establish linkages with external program units to be found in SL's. This last linking step involves only the completion of segment transfer tables. Particular STT entries have already been assigned to particular externals, so no relocation per se need be performed at load time.

PROGRAM files have the following file characteristics:

- Their record length is 128 words.
- They are identified by a file code of 1029.
- The file length is between 4 and 32727 records, inclusive
- The records are fixed length, binary, without carriage control.
- The file consists of only a single extent.

All these restrictions are imposed by the MPE segmenter and loader. Other file configurations are not acceptable.

2.1. Contents of PROGRAM files

A program file is logically divided into five parts, each of which must begin on a record boundary. The parts are as follows:

1. A "fixed area" containing pointers to the rest of the file, and other miscellaneous information
2. An image of the initial global DB area for the program, already fully initialized
3. The code segments comprising the program
4. A list of externals referenced by the program
5. A list of entry points to the program.

These five parts will occur in a PROGRAM file in the order listed. If a program uses no global DB storage, the global DB area part of the file will be omitted. Since all five parts of program files begin on record boundaries, record addresses are used throughout the file for indicating the location of needed information. All record addresses used in program files are single word integers.

2.2. The fixed area

The fixed area occupies the first one or two records of the file. The length of the area, one or two records, depends on whether or not all the information to be included in the area fits in a single record. It will always fit in two. Table 2.2A lists the contents of the fixed area in order.

Table 2.2A -- Fixed Area Contents

Field	Contents
P.(0:1)	program contains fatal error
P.(1:1)	program contains non-fatal error
P.(2:1)	zero the DB area prior to starting execution of the program
P.(3:1)	program contains at least one privileged segment
P.(4:2)	(use undetermined, only zero observed)
P.(6:1)	program has NS capability
P.(7:1)	program has BA capability
P.(8:1)	program has IA capability
P.(9:1)	program has PM capability
P.(10:1)	program has CR capability
P.(11:1)	program has RT capability

Table 2.2A -- Fixed Area Contents (cont.)

Field	Contents
P.(12:1)	program has MR capability
P.(13:1)	(use undetermined, only zero observed)
P.(14:1)	program has DS capability
P.(15:1)	program has PH capability
P(1)	the number of segments in the program
P(2)	the number of words in the global DB area of the program's run-time stack
P(3)	beginning record number of the image of the global DB area of the stack (should be ignored if P(2)=0)
P(4)	beginning record number of the segments list
P(5)	the initial stack size (";STACK=" of prep command)
P(6)	the initial DL size (";DL=" of prep command, zero if ";DL=" not specified)
P(7)	the maxdata specification (";MAXDATA=" of prep command, -1 if ";MAXDATA=" not specified)
P(8)	beginning record number of entry points list
P(9)	logical segment number of the entry points to the program
P(10)	PB relative address of primary entry point
P(11)	execution time DB relative address of a table used by TRACE/3000 (-1 if table not used)
P(12)	execution time DB relative address of the .FORTRAN logical units table, the FLUT (-1 if FLUT not used)

Table 2.2A -- Fixed Area Contents (cont.)

Field	Contents
P(13)	beginning record number of the externals list
P(14)	STT number of primary entry point
P(15)	execution time DB relative address of TRAPCOM', a common block used for interfacing to user trap routines
P(16) to P(27)	these locations have been observed only with the value zero--they are probably reserved, and should always be zero

Following P(27) are two variable length subareas of the fixed area. The first begins in P(28), and is P(1) bytes in length. That is, there is one byte for each segment in the program. This subarea is always an integral number of words in length, so a byte is sometimes wasted on the end. It is believed that the loader uses this subarea for mapping logical segments to actual segments. After a program has been prepared, but before it has ever been loaded, this subarea will contain all zeros.

The second subarea begins in the word immediately following the last word of the first subarea. The second subarea includes a one-word segment descriptor for each segment in the program. These segment descriptors are in the same order as the segments themselves in the file. The format of a segment descriptor is given in table 2.2B.

Table 2.2B -- Segment Descriptor Format

Field	Contents
P.(0:1)	segment is privileged
P.(1:1)	(use undetermined, only zero observed)
P.(2:14)	the length of the segment, in words

2.3. The global DB part

The global DB part is simply an image of all DB and secondary DB which is to be allocated when the program begins execution. As many records as necessary are used to hold the needed number of words. Unused words at the end of the last record, if any, are ignored. If the number of words of DB is given in the fixed area as zero, then this area may be omitted from the file altogether, and the record pointer to this area in the fixed area may be set to one (one is the only value for this pointer yet observed in this context).

This DB area image is fully initialized. It includes TRACE/3000 tables, the FLUT table, common blocks, DB and secondary DB arrays and simple variables, and anything else which must be placed in the DB area. There is no opportunity to add to the DB global area once the program begins execution, since it will be delimited by DB and the initial Q register setting. Allowance must be made for all global run-time storage at the time the PROGRAM file is generated.

2.4. The segments list

The code segments which comprise the program are placed, one after another, in the segments list. Each segment begins on a record boundary, and occupies an integral number of records. The actual length of each segment, in words, is given in the segment descriptor words in the fixed area of the PROGRAM file. Unused words at the end of the last record of a segment, if any, are ignored.

The code segments of a program are often referred to by their logical segment numbers. A logical segment number simply gives the position of the segment in the segments list. The first segment in the segments list is logical segment number zero, the next is logical segment number one, and so on. Actual segment numbers, which will be used in the XCST for the program, are assigned by the loader when the program is loaded. Segment transfer tables will contain the actual segment numbers used when the program was last loaded.

There is a one-to-one correspondence between the entries in the segment descriptor words in the second subarea of the fixed area, and the segments in the segments list. The first descriptor applies to the first segment, the second to the second, and so on. The record number of any

particular segment in the file must be deduced from the segment descriptor words. If, for example, it were desired to find the second segment, the first segment descriptor word would be used to calculate the number of records occupied by the first segment. This number would be added to the record number of the beginning of the segments list given in the fixed area. The resulting record number is the first record of the desired segment.

2.5. The externals list

The externals list includes entries for all externals referenced by the program. For each external it gives the segment and STT numbers of the program segments to be patched with the actual segment and STT numbers of the external. In addition, a parameter information block is included in each entry, which indicates the calling sequence which the program uses to call the external. This list is organized as a simple linear list. The list is terminated by an "entry" with zero in its first word.

Illustration 2.5 shows the format of entries on the externals list. Each entry begins with a name field. The P.(0:4) field of the name field is unused, and should be set to zero. Following the name field is a word with three fields. P.(0:4) of this word should be set to zero. It is unused. The use of P.(4:4) is not fully known, but it appears to be used by the loader to indicate how the external reference was satisfied. When generating the program file, it should be set to zero. P.(8:8) gives the number of references to this external by the program. (This number should never exceed the maximum number of code segments allowed for a single program, since a given external should occur only once in any given segment transfer table.)

Following the word giving the number of references, there is a one-word reference descriptor for each reference. This field is thus variable in length. P.(0:8) of each descriptor gives the segment transfer table entry number for this external in a referencing program segment, and P.(8:8) gives the logical segment number of the referencing program segment. This entire area is shown as a single undivided area in illustration 2.5, but the reader should keep in mind that it is a variable length field.

After the reference descriptors is a parameter information block. This parameter information block is in the same format used in USL files.

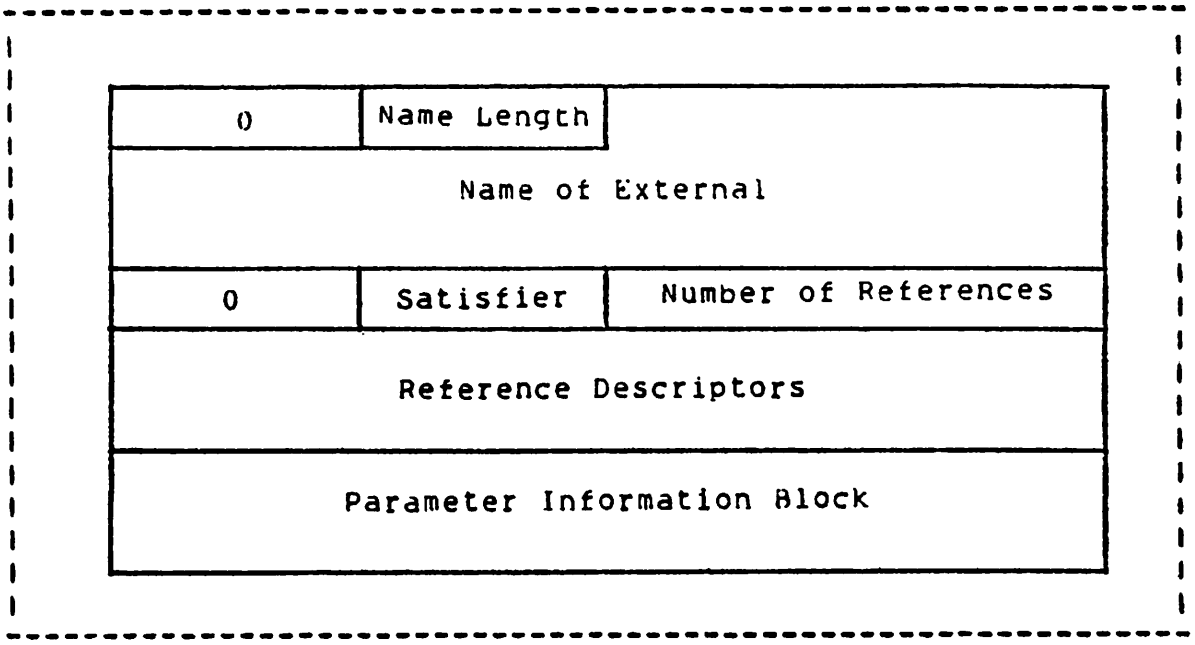

Illustration 2.5 -- Format of Externals List Members

2.6. The entry points list

The entry points list gives all entries to the program. At least the name of the outer block is included in this list. All entry points to the program must of course be in the same segment as the primary entry point. The logical segment number of this segment is indicated in the fixed area in the beginning of the program file. This list is a simple linear list. It is terminated by a list "member" with zero in its first word.

Members of the entry points list all consist of a name field followed by exactly two words. The P.(0:4) field of the name field is unused and should be set to zero. The first of the two words following the name gives the PB relative address of the entry point. The second gives the segment transfer table number of the entry point.

The format of entry points list members is shown in illustration 2.6.

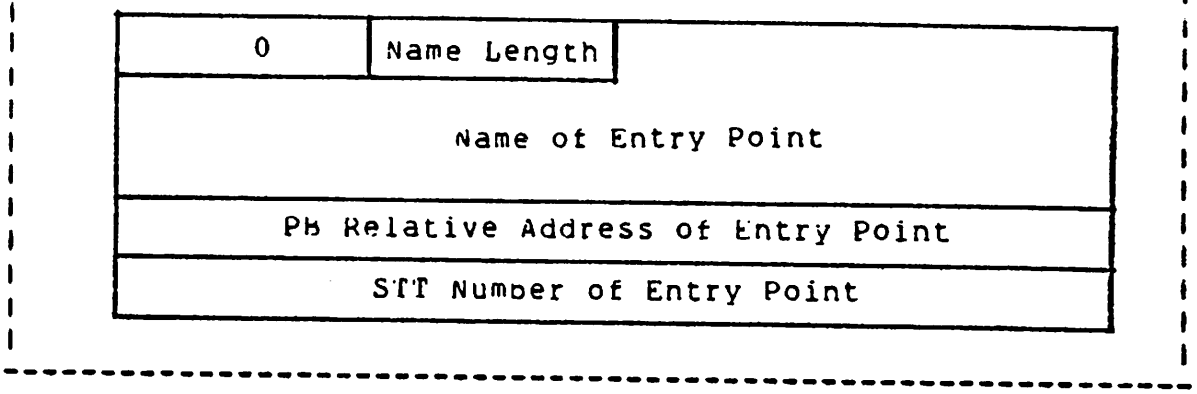

Illustration 2.6 -- Format of Entry Points List Members

3. USL files

USL files are the principal form of input to the MPE segmenter. From a USL file with the appropriate contents the segmenter can generate SL, RL, and PROGRAM files. "Relocatable binary modules" are stored in USL's. Segmentation, global and secondary DB address assignments, external procedure references, PB relative references, and the like have not yet been resolved in these files. Although USL files constitute the most complex form of compiler output on HP3000 computer systems, they are also the most flexible, giving the user the most options.

USL files have the following file characteristics:

- Their record length is 128 words.
- They are identified by a file code of 1024.
- The file length must be from 4 to 32727 records, inclusive.
- Records must be fixed-length binary, without carriage control.

These restrictions are imposed by the MPE segmenter. Unlike program files, USL files may be composed of any number of extents.

3.1. Contents of USL files

A USL file is logically divided into three major parts, each of which must begin on a record boundary. The parts are as follows:

1. "record zero" containing pointers to the rest of the file, list heads, and other miscellaneous information

2. the "directory" containing one entry for every KBM, segment, and entry point in the file

3. the "information block" containing all information headers and code modules.

These three parts always occur in the order specified, and are of a fixed length in any given file. This length may vary from file to file, but within any one file, the boundaries are clearly defined by information in record zero. Record zero is always simply the first record in the USL file.

The directory always begins with the second record of the USL file, at file address 00001 000 (see 3.2 for a discussion of file addressing). The information block always begins at a file address specified in record zero. In both the directory and the information block, information is placed in successive contiguous locations. Record boundaries are not recognized. If any entries or headers are deleted from either of these two areas, the space they formerly occupied is not recovered. It is referred to as "garbage," and is never used again. Never-used directory space is referred to as "available" directory, and never-used information block space as "available" information.

An intrinsic named ADJUSTUSLF, documented in the MPE segmenter reference manual, may be used to expand the directory or the information block as desired. The directory, however, may not exceed 32K words, and the file may not exceed a total of 32K-1 records.

A USL file may be initialized to the empty state via the intrinsic INITUSL. This intrinsic is documented in the MPE segmenter reference manual.

In total, then, a USL file consists of the following five parts, in this order: record zero, in-use directory, available directory, in-use information, and available information. These areas are delimited by pointers and length values kept in record zero.

3.2. File addressing

It is appropriate at this point to discuss USL file addressing. Locations within a USL file are identified by specifying the word within the file which is of interest. The first word in the file is word number zero. Seven bits are required to specify the offset of a word from the beginning of a record. Up to 15 bits may be required to specify record within file. Full file addresses are thus normally stored in double words, in which strictly speaking only the low order 22 bits are significant.

File addressing within the directory is somewhat different. Single word addresses are used. P.(9:7) is still a word offset into a particular record, but P.(1:8) is now the record number.

Unused high-order bits in both double and single word file addresses should be set to zero. This will provide compatibility with the MPE segmenter.

The high order bit of a file address often has special significance. It may be used to indicate that the address is a thread link instead of a normal link. It may be used to indicate the end of a list. It has various uses, which should be carefully considered when interpreting any file address.

The address zero has special significance. It is the null address. No pointer ever points to record zero.

File addresses are often relative to some location in the file. The starting address of the information block is normally used as the base address. The value of a relative address (even if it is zero) is added to the base address to obtain an actual file address. The high-order bit of an address should not be involved in this calculation, since it has special interpretations. It should be set to zero in both the base and the offset before adding. It is essential always to consider whether an address is absolute or relative. This depends on the context in which the address occurs.

In symbolic form, file addresses are represented by two octal numbers. The first is five digits in length, and specifies the record number. The second is three digits in length, and specifies the offset to the desired word in the record. These two numbers are separated by a blank. If the high-order bit of an address is on, then '(1)' is prepended to the five-digit record number. (If in the given context the high-order bit has no significance, the '(1)' may be omitted.)

3.3. Record zero

Record zero occupies the first record of a USL file. It contains pointers and counters essential to interpreting the remainder of the file. Table 3.3 lists the contents of record zero in order. A name is given to each field which is used as a convenience in referring to the field.

Table 3.3 -- USL File Record Zero Contents

Word(s)	Name	Contents
0		the constant '1' -- apparently identifies the file as a valid USL file
1	NDE	number of entries in the directory
2	DL	the directory length, in words (number of words which have already been allocated; includes garbage)
3	DG	number of words of DL which are 'garbage'
4	NDGE	number of directory garbage entries
5	BDL	list head for the block data list
6	IPL	list head for the interrupt procedure list
7	SL	list head for the segments list
8,9	FL	length of the entire USL file, in words
10	SAAD	start address of available directory space (should be equal to 00001 000 + DL)
11	ADL	available directory length, in words

Table 3.3 -- USL File Record Zero Contents (cont.)

word(s)	Name	Contents
12,13	SAIB	start address of the information block (should be equal to SAAD + ADL)
14,15	IBL	length of the information block, in words (number of words which have already been allocated; includes garbage)
16,17	SAAIB	starting address of the available information block space
18,19	AIBL	length in words of the available information block
20,21	IBG	number of words of IBL which are 'garbage'
22	NIBGE	number of garbage information block entries
23-32		apparently reserved; should always be set to zero
33-127		hash list heads

The use of many of these fields will be explained in subsequent sections of this paper. The use of others is indicated in illustration 3.3. This illustration labels various locations and areas in a USL file with the names assigned to the fields of record zero which refer to those portions of the USL file.

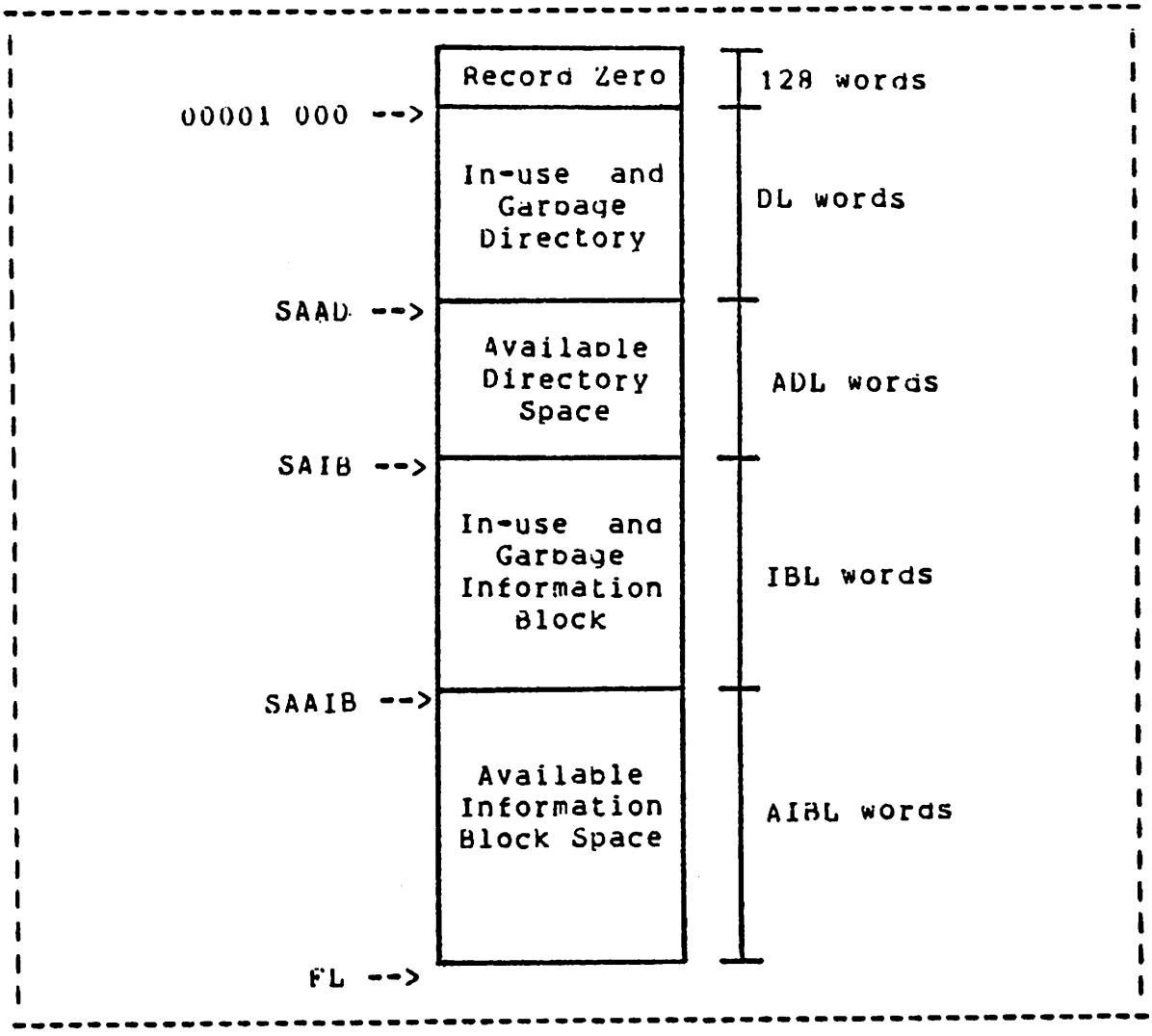

Illustration 3.3 -- Use of Record Zero Fields

3.4. The directory

The USL file directory contains all information needed to process and manipulate information contained in the information block. A great variety of items are found in the directory, but they are easily classified into distinct groups. The elementary directory data item is the 'entry.' The data structures formed from entries are familiar sorts of trees and linked lists. Several implementation aspects of the directory, which do not conveniently fall into any other section, are listed below:

1. The directory begins at file location 00001 000.

2. Each directory entry consists of some number of contiguous words. Entries are not, however, necessarily contiguous within the directory. That is, there may be some unused space between entries.

3. File record boundaries are not recognized within the directory. Entries may span record boundaries freely.

4. All pointers to entries in the directory are absolute pointers. If the pointer is contained within record zero, or within the directory itself, it is a single word, and not a double word, pointer. (See section 3.2.)

5. All pointers to the information block which are contained in the directory are double word pointers. All such pointers are relative to SAIB. (See sections 3.2 and 3.3.)

The entries contained in the directory are related on lists. There are only eight types of entries, and only four types of lists, all of which are described in detail in the following subsections.

3.4.1. Directory lists

The four types of lists in the directory are as follows: the interrupt procedure list, the segment list, the block data list, and the hash lists. Each of these four is discussed in a separate subsection below. With each subsection is provided an illustration of the list discussed.

For every list in the directory, there is a certain type of entry, or there are certain types of entries, which are included on that list. With the exception of the hash lists, only entry types appropriate to the list may be placed on the list. All entry types are appropriate to the hash lists. Entries may be included only on lists for which they are appropriate.

Every entry is on exactly two directory lists. It is on one, and only one, of the block data list, the interrupt procedure list, and the segment list. It is also on one, and only one, of the hash lists.

3.4.1.1. The interrupt procedures list

The interrupt procedure list ('IPL') is a linear linked list. Its list head is IPL in record zero (see 3.3). All interrupt procedure directory entries are on this list. Entries are linked together by their brother pointers, with a link of zero terminating the list. As of this writing, no further information is available about the IPL directory list.

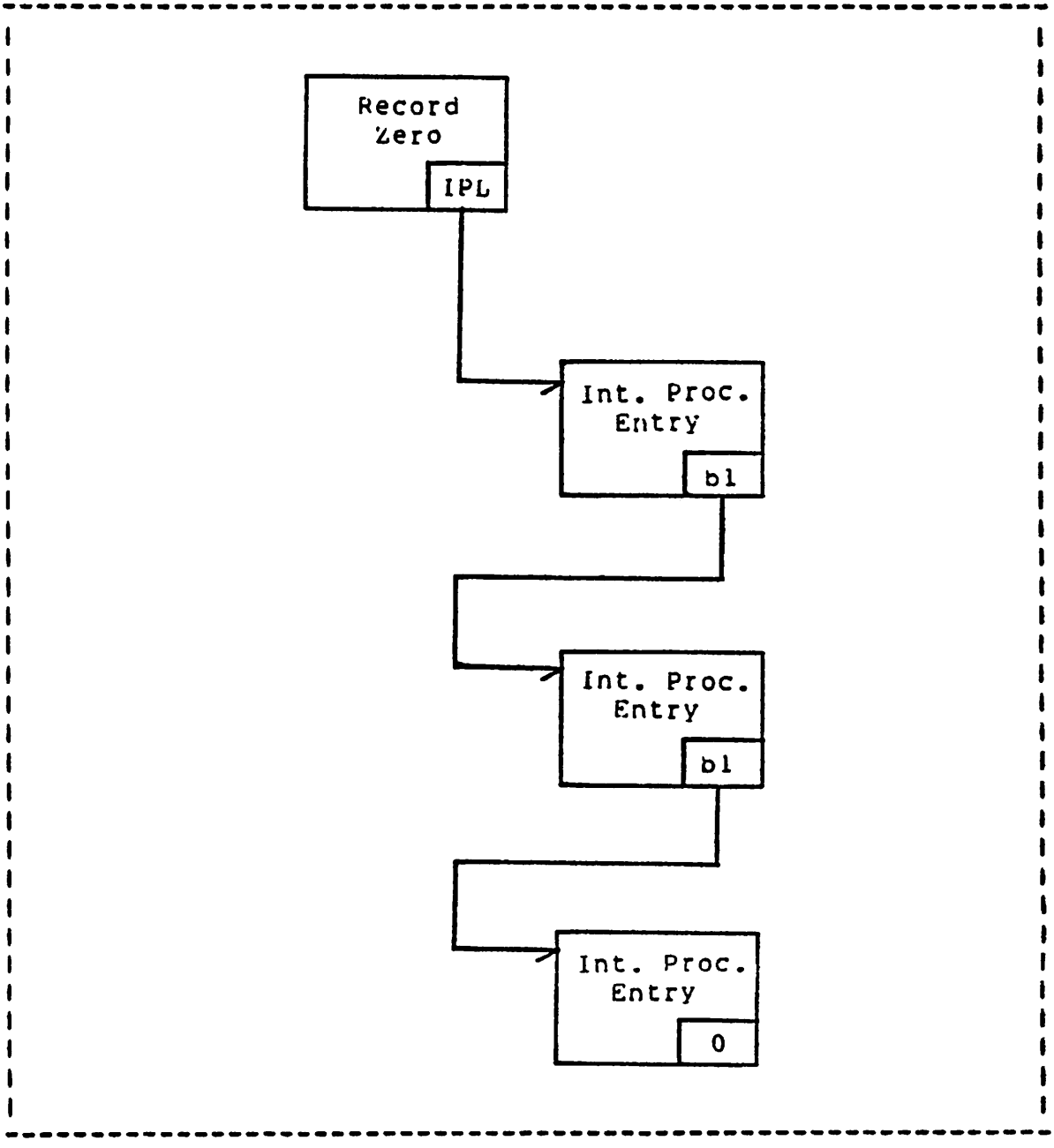

Illustration 3.4.1.1 -- Interrupt Procedures List

3.4.1.2. The block data list

The block data list ('BDL') is a linear linked list. Its list head is BDL in record zero (see 3.2). Block data subprograms generate block data RBM's. All block data directory entries are on the BDL directory list. They are linked together by their brother pointers, with a link of zero terminating the list.

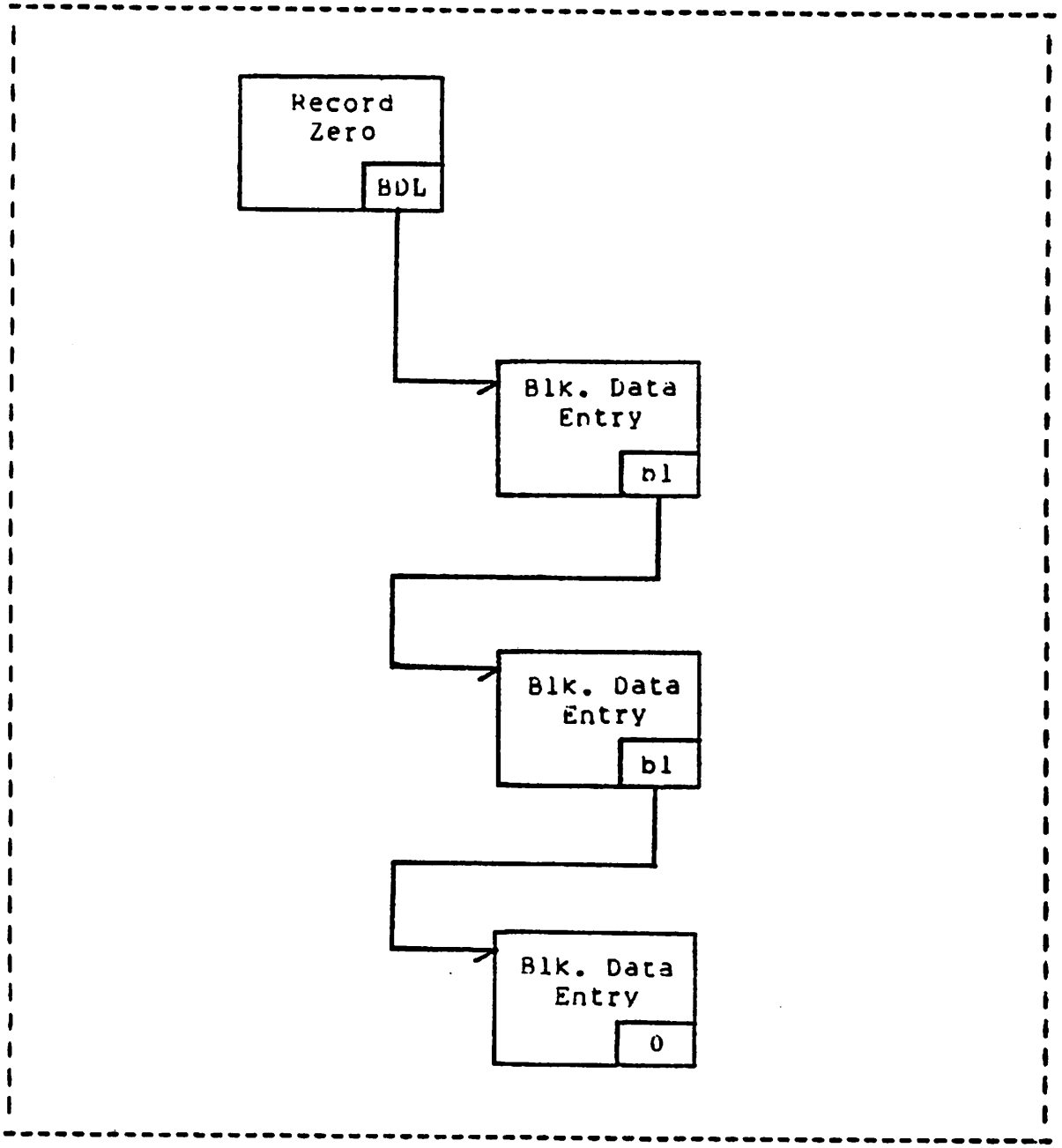

Illustration 3.4.1.2 -- Block Data List

3.4.1.3. The segment list

The segment list is really a tree. The pointer to the root of the tree is SL in record zero (see 3.2). The following three types of entries are found in the tree: segment entries, RBM entries (which may be either primary outer block or primary procedure type entries), and secondary entry point entries (which may be secondary outer block, secondary procedure with parameters, or secondary procedure without parameters type entries).

The segment entries do actually form a linear linked list, with SL in record zero as the list head. They are linked by their brother pointers, with a link equal to zero terminating the list.

A segment entry may have zero or more sons. The immediate sons of a segment entry must be RBM entries. The son pointer of a segment entry points to the segment entry's first son. All the sons of a given segment are linked in a linear list by their brother pointers. This family list is terminated by a link with its high-order bit turned on, and which points back to the parent segment entry. For example, if a segment entry is located at 00033 027, then the link terminating the family list would be (1)00033 027. If this segment entry had no sons, then the segment entry's son pointer would be (1)00033 027.

Each RBM entry may also have zero or more sons. The sons of an RBM entry are secondary entry point entries. The son pointer of an RBM entry is analogous to the son pointer of a segment entry. It points to the first son, and other sons are linked into the family by their brother pointers. Again, the family list is terminated by a link with its high-order bit turned on, and which points back to the parent RBM entry. If an RBM entry has no sons, then its son pointer points to itself, just as in the case of segment entries with no sons.

The links terminating family lists in the segment list are referred to as 'thread' links, since they refer back to the root of the subtree in which a node is located. The son relationship is defined only from segment to RBM entries, and from RBM to secondary entry point entries. The brother relationship is defined from segment to segment, from RBM to RBM, and from secondary entry point to secondary entry point entries. The father relationship is defined from RBM to segment, and from secondary entry point to RBM entries. The tree is thus a left-son/right-sibling, threaded tree data structure.

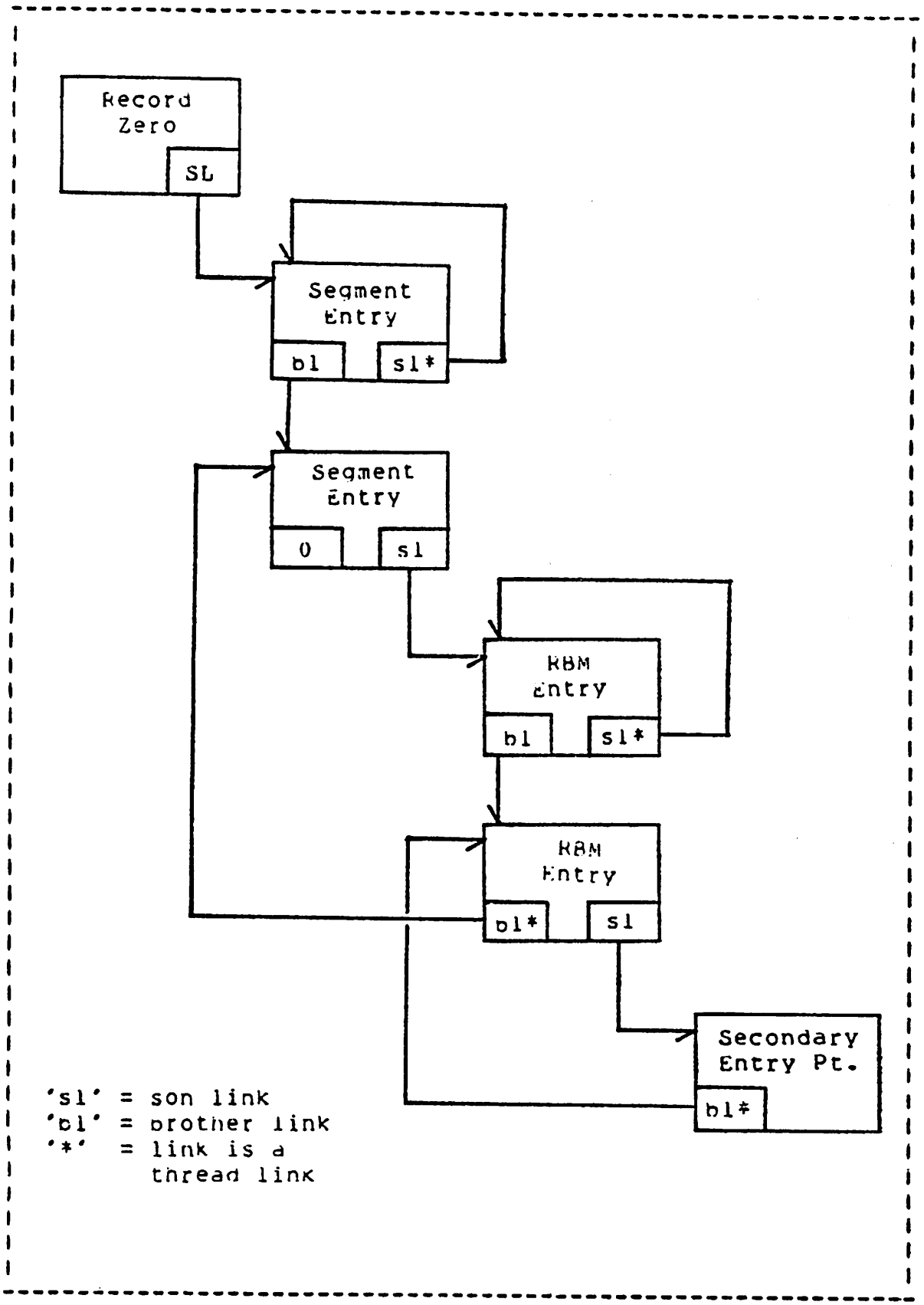

Illustration 3.4.1.3 -- Segment List

3.4.1.4. The hash lists

To facilitate quick access to directory entries by name, every directory entry is also placed on a hash list. USL files use 95 hash lists, the list heads of which are in record zero (see 3.2). Each list head is a single word absolute pointer into the directory. Directory entries which hash to the same list are linked to each other by their hash links (see 3.4.2). Each of the 95 hash lists is thus a linear linked list. A zero link terminates a hash list. Hashing a name produces an integer between 0 and 94, inclusive, which is used as an index into the 95 hash list heads to access the hash list on which the entry referred to by the name is located.

An entry should always be added to a hash list nearest to the hash list head. That is, the hash list head should be made to point to the entry, and the entry's hash link to point to the entry formerly pointed to by the hash list head.

The MPE segmenter refers to the 'index' of an RBM, or directory entry. This is a reference to how recently the entry was added to its hash list. The most-recently added is indexed one, the next-most-recently is indexed two, and so on. "Least-recent" on any given list refers to the entry on the list with a hash link equal to zero. When the MPE segmenter refers to the index of an entry, only the entries of a given name are considered. The entire list is not relevant. The index zero has a special meaning. It refers to the most-recent active entry having the given name on the hash list (active/inactive entries are discussed in the MPE segmenter reference manual).

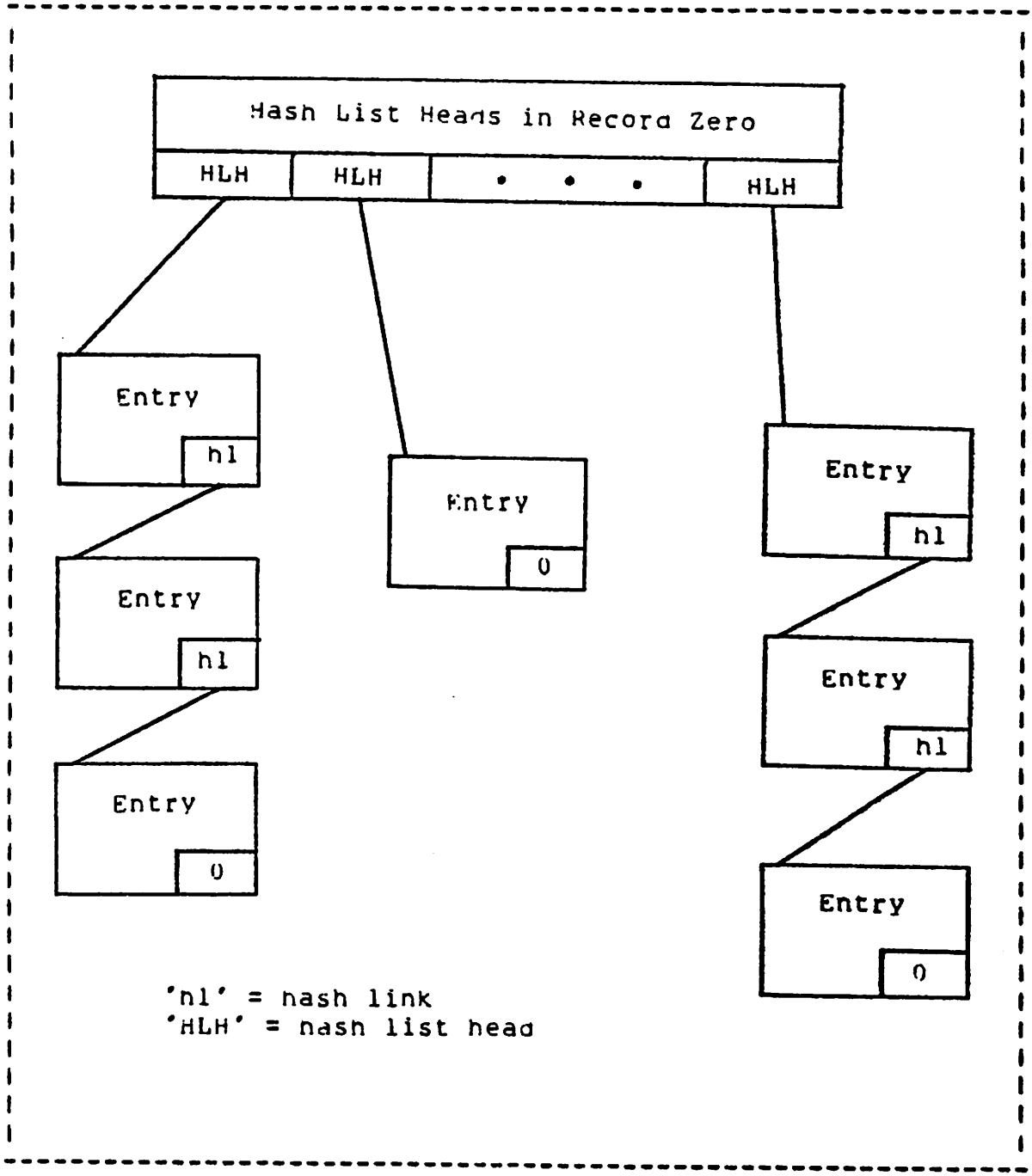

Illustration 3.4.1.4 -- Hash Lists

3.4.2. Directory entries

In this section, all eight entry types found in the directory are presented. All entries have some fields in common, which together form a standard directory entry prefix. Included in this prefix is a name field, giving the name associated with the entry. 'P1' is used to denote an integer pointer which points to the word immediately following the last word used by the name field. Illustration 3.4.2 shows the layout of the prefix. The contents of the prefix are described in table 3.4.2A.

Table 3.4.2A -- Directory Entry Prefix Contents

Field	Contents
P(0).(1:10)	number of words in this entry
P(0).(11:5)	type of this entry; types have following designations: 1 segment entry 2 primary outer block entry 3 secondary outer block entry 4 primary procedure entry 5 secondary procedure entry, without parameters 6 interrupt procedure entries 7 block data entry 8 secondary procedure entry, with parameters Other entry types are undefined
P(1)	the entry's hash link (see 3.4.1.4)
P(2)	the entry's name field
P1(0)	the entry's brother link

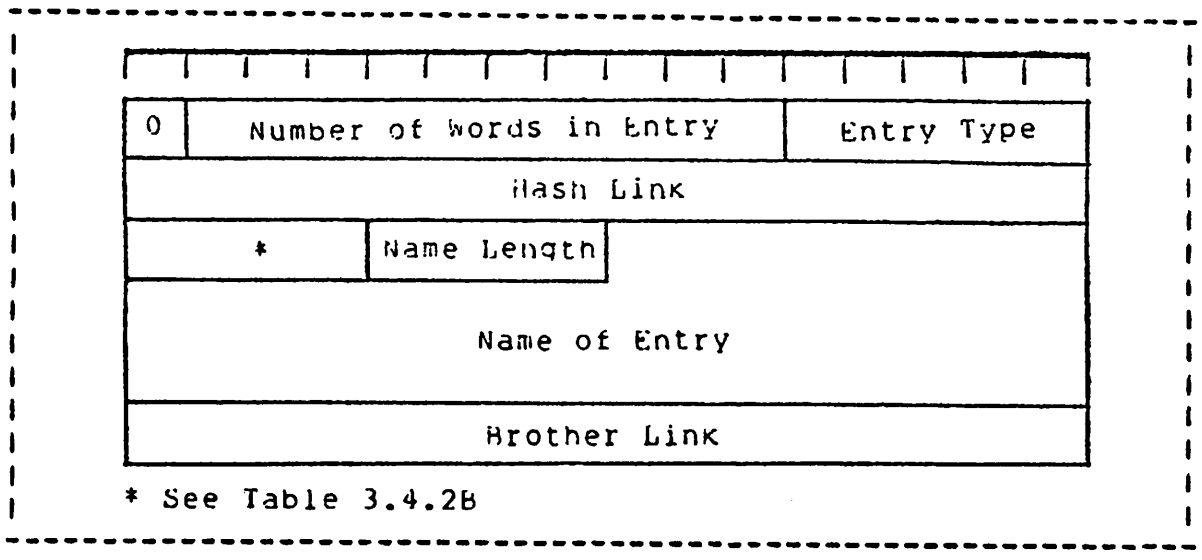

Illustration 3.4.2 -- Standard Prefix

The P(2).(0:4) field (that is, the first nibble of the name field) has important uses. The interpretation of this field depends on entry type. These interpretations are given in table 3.4.2B.

Table 3.4.2B -- Interpretation of P(2).(0:4)

Entry	Field	Interpretation
1	0:1	shows whether entry is active or not ('1'=inactive)
	1:3	reserved; should be set to zero
2	0:1	shows whether entry is active or not ('1'=inactive)
	1:1	shows whether entry is callable ('1'=uncallable)
	2:1	shows whether program unit must execute in privileged mode
	3:1	reserved; should be set to zero

Table 3.4.2B - Interpretation of P(2).(0:4) (cont.)

Entry	Field	Interpretation
3	0:1	shows whether entry is active or not ('1'=inactive)
	1:1	shows whether entry is callable ('1'=uncallable)
	2:2	reserved; snould be set to zero
4	0:1	shows whether entry is active or not ('1'=inactive)
	1:1	shows whetner entry is callable ('1'=uncallable)
	2:1	shows whetner program unit must execute in privileged mode
	3:1	shows whether entry is hidden
5	0:1	shows whether entry is active or not ('1'=inactive)
	1:1	shows whetner entry is callable ('1'=uncallable)
	2:1	reserved; should be set to zero
	3:1	shows whether entry is hidden
6	0:1	shows whether entry is active or not ('1'=inactive)
	1:2	apparently an interrupt procedure type number
	3:1	reserved; should be set to zero

Table 3.4.2B - Interpretation of P(2).(0:4) (cont.)

Entry	Field	Interpretation
7	0:1	shows whether entry is active or not ('1'=inactive)
	1:1	set if fatal error in block data RBM
	2:1	set if non-fatal error in block data RBM
	3:1	reserved; should be set to zero
8	0:1	shows whether entry is active or not ('1'=inactive)
	1:1	shows whether entry is callable ('1'=uncallable)
	2:1	reserved; should be set to zero
	3:1	shows whether entry is hidden

In the subsections of this section all entry types are explained. A diagram of most types is presented to illustrate the format of the entry. Mention will often be made of "parameter information blocks," "header information blocks," and "header information sets" ('PIB', 'HIB', and 'HIS', respectively). HIB and HIS are described in 3.4.3, and so are not further discussed here. A PIB provides the calling sequence of a program unit. It will always in illustrations be drawn as a single large area, but the reader should keep in mind that it is really comprised of one or more words, and is thus a variable length field. 'P2' is used to denote an integer pointer pointing to the word immediately following the PIB.

3.4.2.1. Segment entries

For segment type directory entries, only a single word is appended to the standard prefix. That word contains the son link of the entry. (The significance of a son link is discussed in 3.4.1.)

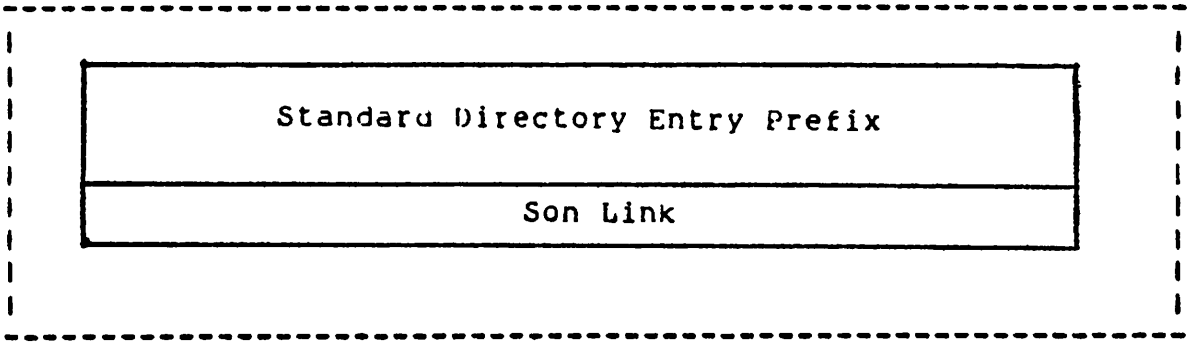

Illustration 3.4.2.1 -- Segment Entry

3.4.2.2. Primary outer block entries

A number of fields follow the standard prefix in primary outer block type directory entries. They are described in table 3.4.2.2. In this table, 'P' is assumed to be an integer pointer to the word of the entry immediately following the standard prefix. Each field is given a name to simplify reference to the field in this paper.

Table 3.4.2.2 -- Primary Outer Block Fields

Field	Name	Contents
P(0)	SONL	son link of the entry
P(1)	PUSA	program unit starting address (address within the code module of the entry point)
P(2),P(3)	SAC	starting information block file address of the code module (this address is relative to SAIB; see section 3.3)
P(4).(0:1)	ERROR	set if program unit contains a fatal error
P(4).(1:1)	WARN	set if program unit contains a non-fatal error

Table 3.4.2.7 -- Primary Outer Block Fields (cont.)

Field	Name	Contents
P(4).(2:14)	CODELEN	number of words in the object code module
P(5)	STACKEST	an estimate of the number of words of stack needed by the program unit
P(6)	PDB	the number of words of primary DB allocated by this program unit
P(7)	SDB	the number of words of secondary DB allocated by this program unit
P(8)	TPCLEN	number of words in a table used by TRACE/3000
P(9)	DATALEN	number of words in secondary DB reserved by DATA (FORTRAN) or OWN (SPL) declarations
p(10) to end of entry		header information block for the program unit

SDB and DATALEN are not the same thing. DATALEN refers to the number of words in a 'secondary DB array. Associated with each program unit is an area of secondary DB space. This area includes such things as the FORTRAN logical units table, format strings (those referenced in a read statement and containing 'H' specifications must be globally located, to retain their values), own/data variables, and the like. Included in DATALEN is only that amount of storage to be allocated for own/data variables--this portion of the secondary DB storage allocated by a program unit is referred to as the secondary DB array. The following are not included in DATALEN or SDB: the FORTRAN logical units table, TRACE/3000 tables, and common arrays. SDB does include the number of words used by globally located format strings.

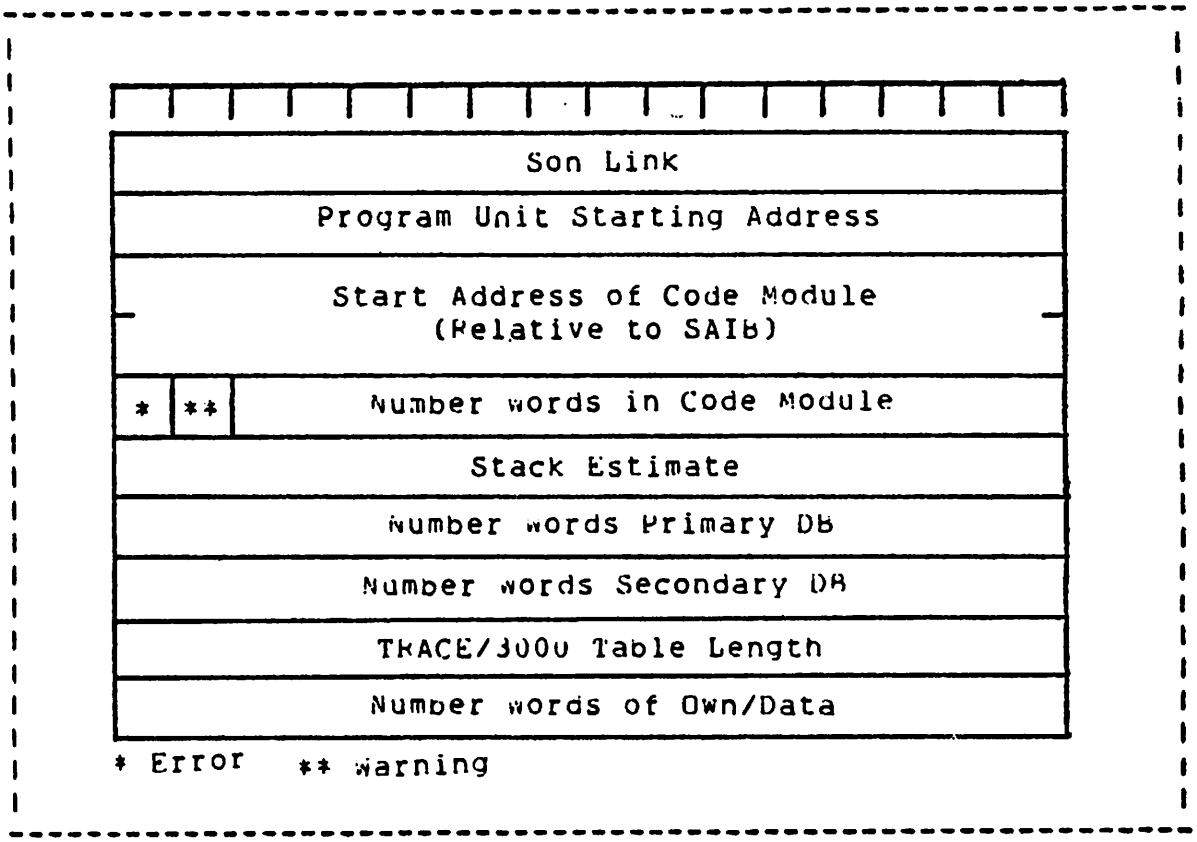

Illustration 3.4.2.2A - O.B. and Proc. Entries Body

The first ten words of this entry are shown in illustration 3.4.2.2A. The format of primary block entries as a whole is shown in illustration 3.4.2.2B.

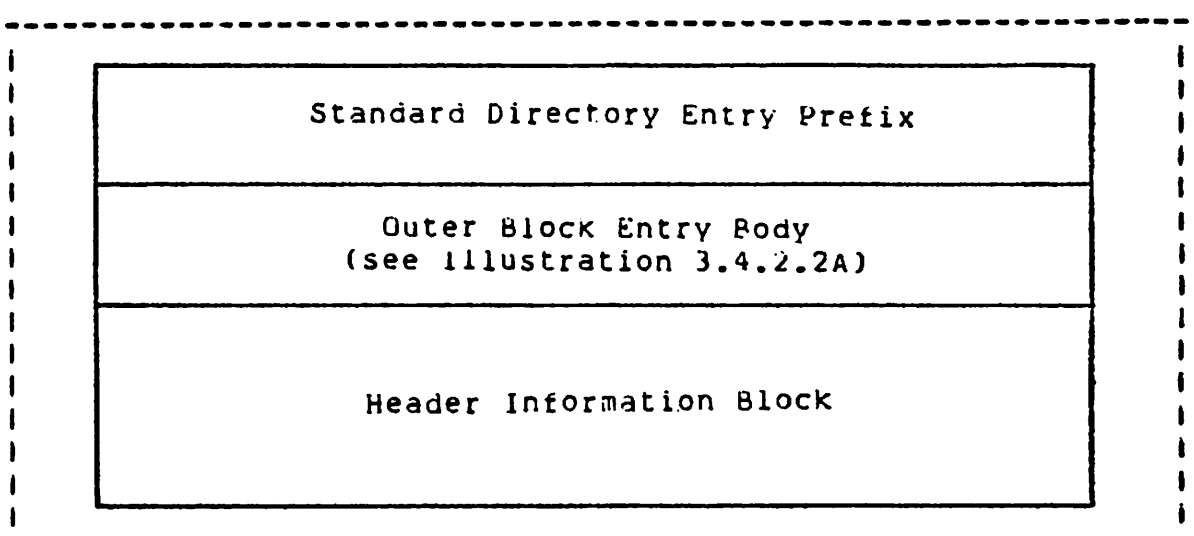

Illustration 3.4.2.2B -- Primary Outer Block Entry

3.4.2.3. Secondary outer block entries

Only a single word is appended to the standard prefix for secondary outer block type directory entries. This word indicates the location of the entry point in the code module associated with the parent directory entry. The word is given as a word-offset from the beginning of the code module, and so is analogous to PUSA, described in 3.4.2.2.

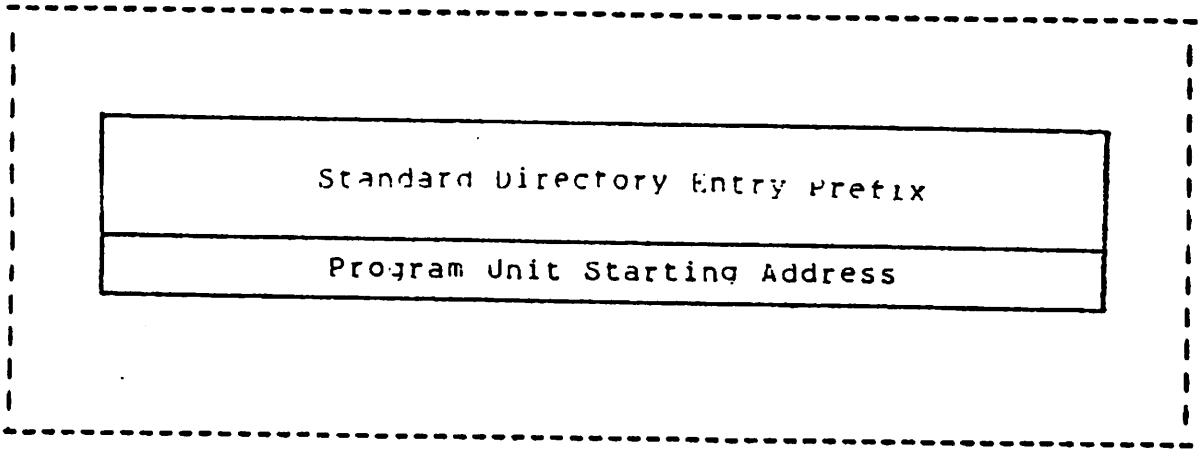

Illustration 3.4.2.3 -- Secondary Outer Block Entry

3.4.2.4. Primary procedure entries

Primary procedure type directory entries append to the standard prefix exactly the same information as is appended by primary outer block entries, with one exception. Between DATALEN and the header information block, a parameter information block is inserted.

3.4.2.5. Secondary procedure without parameters entries

The secondary procedure without parameters directory entry type appends to the standard prefix only a single word. This word contains the address of the entry point to the code module associated with the parent entry. The address is a word-offset from the beginning of the code module, and so is analogous to PUSA, described in 3.4.2.2.

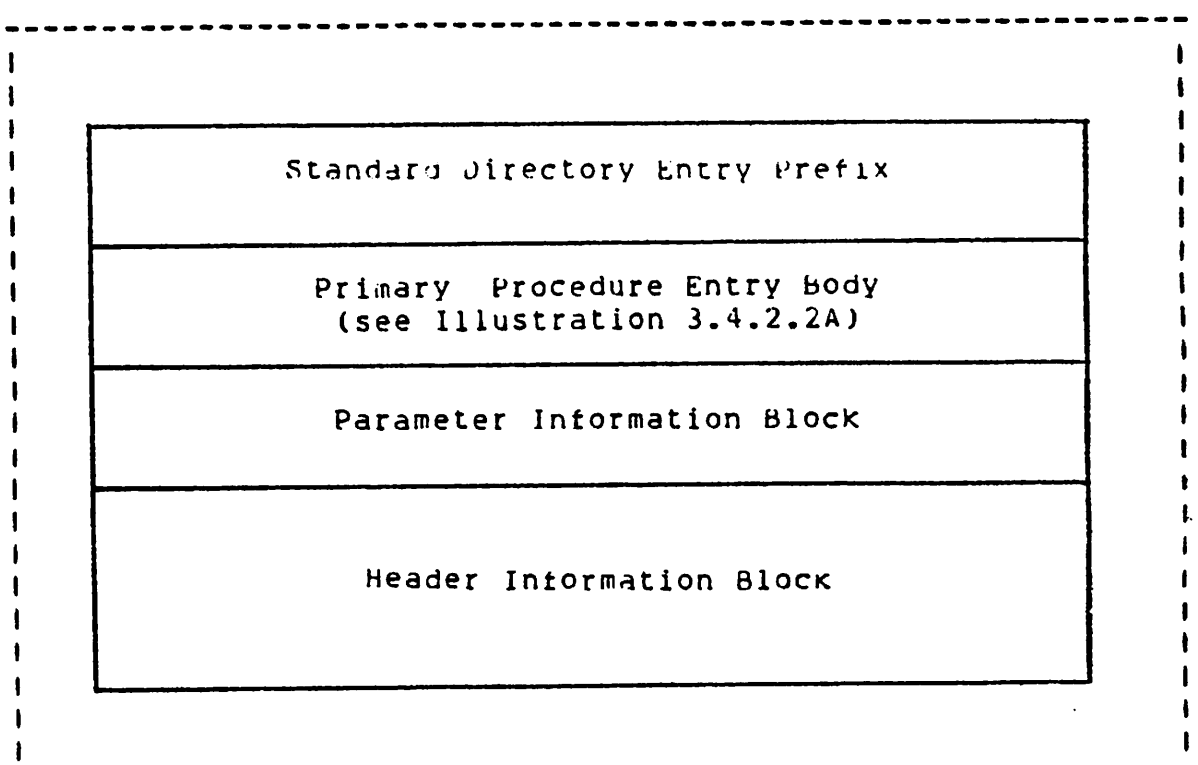

Illustration 3.4.2.4 -- Primary Procedure Entry

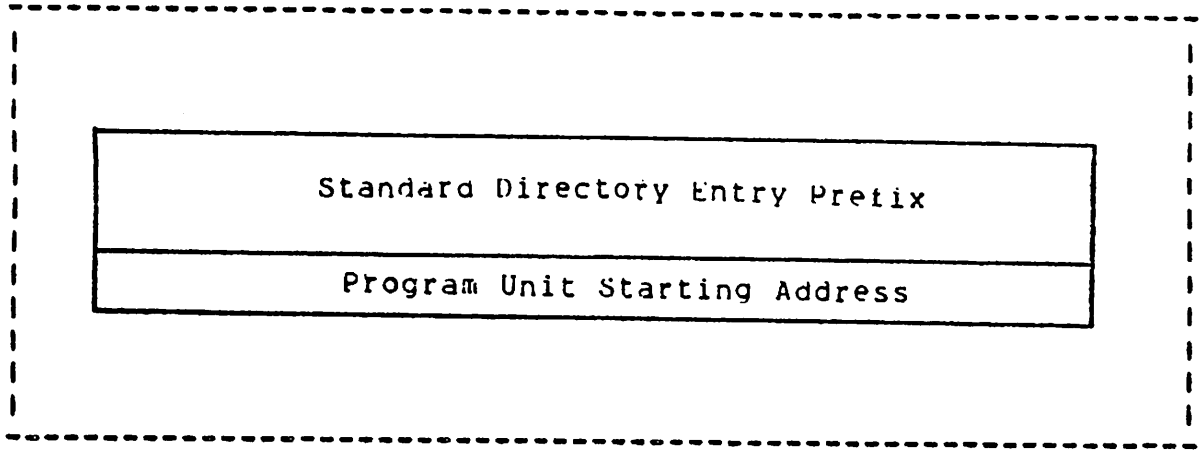

Illustration 3.4.2.5 -- Sec. Procedure, No Parms., Entry

3.4.2.6. Interrupt procedure entries

After the standard prefix, interrupt procedure type directory entries append five words. These five words are followed by a header information block. The proper interpretation of the five words has not yet been determined.

3.4.2.7. Block data entries

A block data type directory entry appends after the standard prefix a number of subentries. Each subentry contains information for one block of common. (As far as the segmenter is concerned, every block of common is named. The name "CUM'" is used to refer to blank common.) There is no explicit indication of the number of subentries present. This must be deduced from the subentries themselves, and from the number of words in the entry as a whole.

The first word of a subentry gives the number of words in the common block. Following this is a name field giving the name of the common block. Beginning in the word immediately following the name field, there is a header information block for the subentry. Thus, in a manner of speaking, the common block name and length are prepended to the relevant header information block.

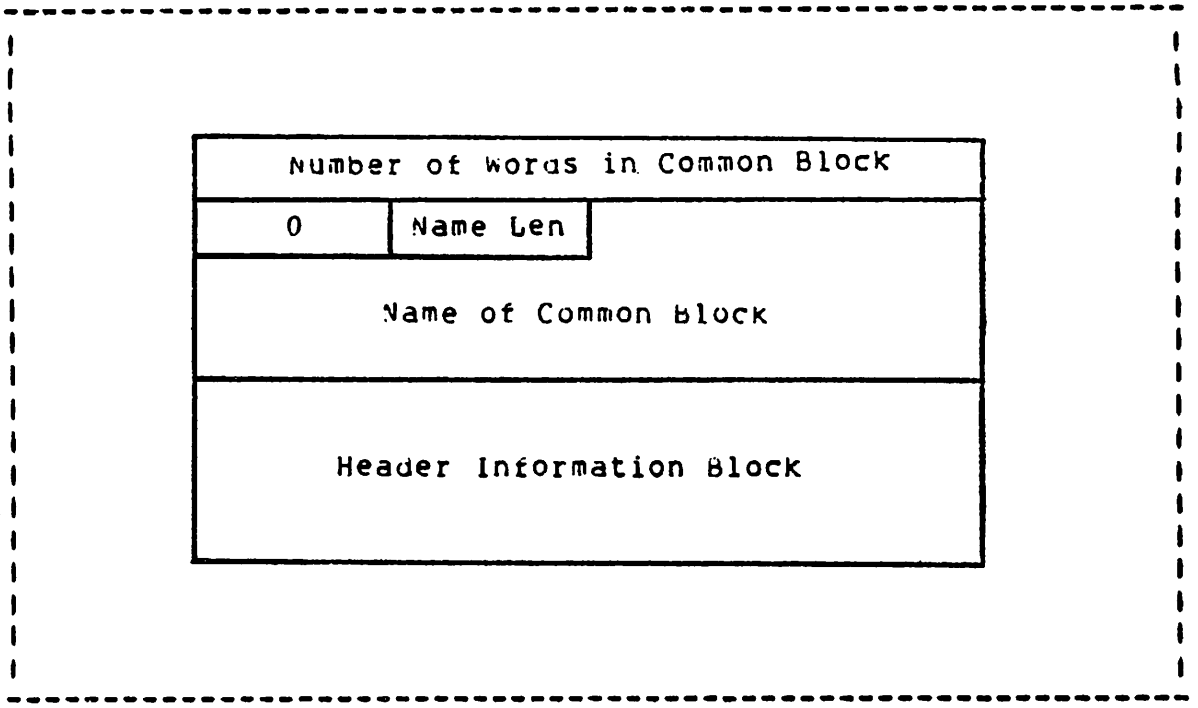

Illustration 3.2.4.7A -- Block Data Subentry Format

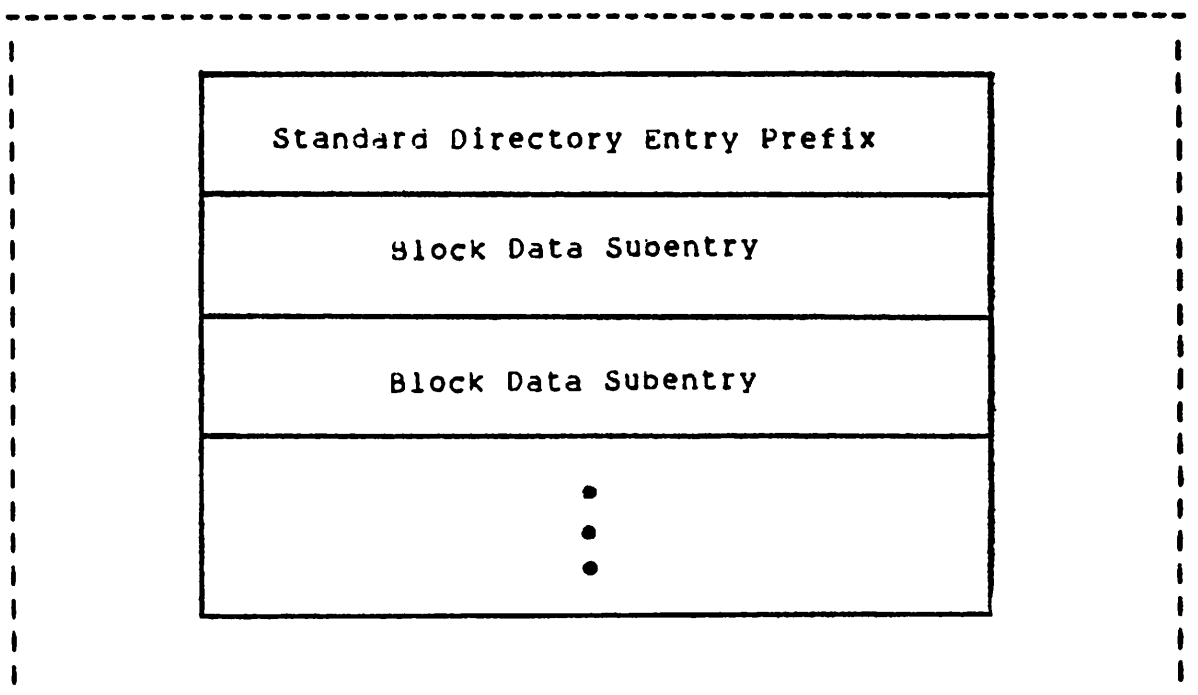

Illustration 3.2.4.7B -- Block Data Entry Format

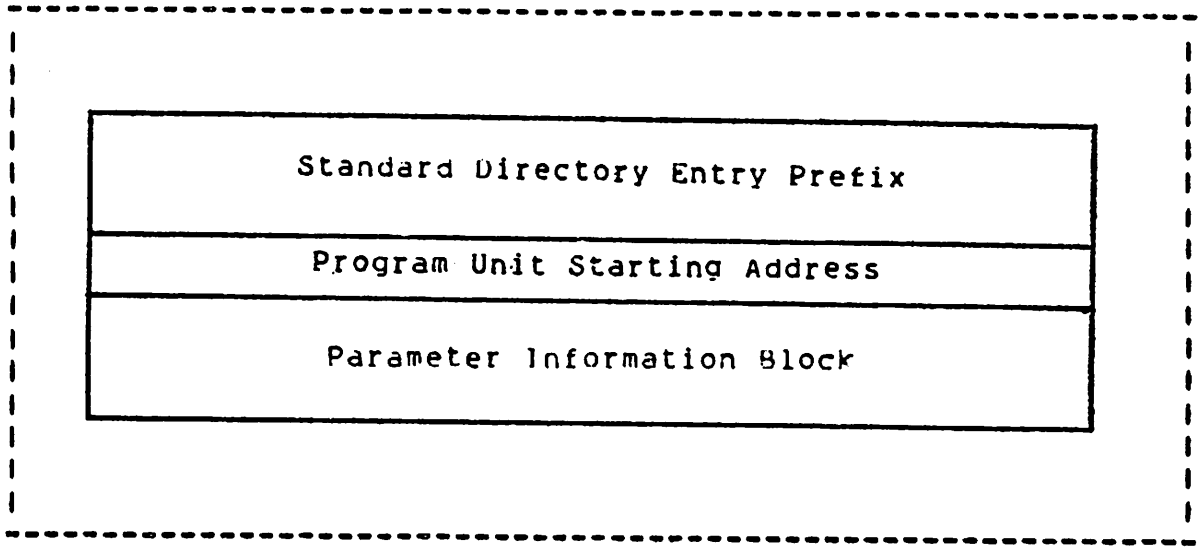

Illustration 3.4.2.8 -- Sec. Procedure, with Parms. Entry

3.4.2.8. Secondary procedure with parameters entries

One word, and then a parameter information block, are appended to the standard prefix in secondary procedure with parameters type directory entries. The word placed between the prefix and the PIB contains the address of the entry point to the code module associated with the parent entry. This address is given as a word-offset from the beginning of the code module.

3.4.3. Header information blocks

In a USL file in MPE, a 'header' is an entry in the information block of the file which provides information necessary for relocating a program unit, and for binding it with other program units. The various types of headers which are possible are discussed in section 3.5. In this section, the header information blocks found in directory entries are discussed. HIB's are used to provide information about the number, types, and lengths of headers associated with a directory entry.

A header information block is divided into header information sets. Each HIB is a more or less distinct entity. Any number of header information sets (including zero) may be included in any header information block. There is no explicit indication of the number of header

information sets are in a header information block. This must be deduced from the HIB itself, and from the number of words in the directory entry as a whole. A HIS begins with a word which specifies in its (1:15) field the number of 'header descriptor words' that are present in the HIS. The (0:1) field of this word is set to zero unless this HIS is the last HIS of the HIB. In this case, (0:1) is set to one. This word is followed by a double word file address which is relative to SAIB (see 3.3). This address points to the first word of the of the first of the actual headers corresponding to the HIS. (All headers corresponding to a given HIS must be contiguous in the information block. See section 3.5.)

In the third and following words of a HIS are header descriptor words. There is one descriptor word for each header associated with the HIS, and the descriptors are in the same order as the headers themselves in the information block. A header descriptor has only three fields. The first, (0:1), is unused, and should be set to zero. The second, (1:10), gives the length of the associated header in words. (The length of headers is thus limited to a maximum of 1023 words.) The third, (11:5), gives the number of the type of the header. Header type numbers are presented in section 3.5.

3.5. The information area

The information block in a USL file contains all header entries. All addresses in the directory which refer to the information block are relative to SAIB (see 3.3). Because of this, the entire information block may be moved up and down in the file, changing only a few fields of record zero. Record boundaries are not recognized in the information block, but it should nonetheless begin on a record boundary.

3.5.1. Code modules

A code module is a special sort of header. It has no associated header descriptor word, it may be longer than the normal maximum of 1023 words, and it is never explicitly included in any HIS (see 3.4.3). It may, however, be placed anywhere within the headers associated with a HIS. The starting address of the code module, SAC

(see 3.4.2) must be used to detect the presence of the code while sequentially processing the headers. If the code module is not needed when detected, it may simply be skipped. It is, as are all headers of a single HIS, contiguous with both the preceding (if any) and following (if any) headers.

The code module contains for the most part finished code, ready to be placed into a program. There can be many exceptions to this, however, depending on other headers associated with directory entries associated with the code module. There are various linear lists and relocatable addresses in the code module itself which are used by these other headers. The relevant lists and addresses will be discussed below together with the appropriate headers.

3.5.2. Information headers

There are twelve types of information headers. They are numbered as follows:

0 null (a garbage header)
1 PCAL, LLBL, or program unit PB address
2 PB address
3 own/data variable (for address correction)
4 secondary DB initializations
5 a table for TRACE/3000
6 variables declared GLOBAL
7 variables declared EXTERNAL
8 primary DB declarations and initializations
9 common (accomplishes only address correction)
10 FORTRAN logical units
11 globally located formats

These numbers are used in header descriptor words. (Header descriptor words were introduced in section 3.4.3.) Every header begins with a header descriptor word which describes it. The format of these descriptors is as follows:

(0:1) reserved, should be set to zero
(1:10) the length of the header in words
(11:5) the number of the header type

Although all headers begin with a descriptor word, each is thereafter highly individual. Each type is described in a separate subsection below.

All of the headers associated with a given HIS must be contiguous within the information block. The directory gives only the file address of the first word of the first header of any HIS. If the headers are not contiguous, it will not be possible to locate them in the file.

3.5.2.0. Null headers

A null header is a garbage entry. It simply takes up as much space as indicated in the header descriptor word. It has no significance to the program unit with which it is associated.

3.5.2.1. PCAL headers

PCAL headers provide all information needed to link the program unit to external program units. It actually has three functions, as follows: to make PCAL patches, to make LLBL patches, and to make procedure PB relative address patches. It is structured as indicated in table 3.5.2.1.

Table 3.5.2.1 -- PCAL Headers

Field	Contents
P(0)	header descriptor word
P(1)	word offset into code module to the first word of a linked list of references to the program unit described in the header; each word in the list in the code module has the following format: .(0:1) 0=patch in a PCAL instruction, 1=patch in an LLBL instruction .(1:1) 0=patch as indicated by .(0:1) 1=patch in PB relative address of the program unit .(2:14) link to next list item (this is a self-relative backwards pointer; the list terminates with a zero pointer)
P(2)	a name field, giving the name of the external program unit (P(2).(0:4) is unused, and should be set to zero)
P1	following the name field is a parameter information block

3.5.2.2. PB address headers

This header provides a means of patching words in the program unit which contain PB relative addresses. After the header descriptor word, the header is simply a series of pointers, each of which is a word-offset into the code module. (The number of these pointers must be deduced from the length of the header as a whole.) In each word in the code module thus pointed to, the compiler must place a PB relative address. This address will be corrected by the MPE segmenter at prepare time by adding to it the PB relative address of the first word of the program unit.

3.5.2.3. Own/data headers

At compile time, the run time address of an own or data variable is not known. It is assigned at prepare time. The MPE segmenter solves this problem by requiring the compiler to place in the code module a pointer to the variable. This pointer will of course then be part of the code at run time. The compiler initializes this pointer to the offset into the program unit's secondary DB array assigned by the compiler to the variable. At prepare time, the segmenter will add to this value the DB offset of the program unit's secondary DB array, thereby providing the code at run time with the correct pointer value.

After the header descriptor word, the entire header consists of pointers each of which is a word-offset into the code module. (The number of pointers must be deduced from the length of the header.) Each points to a location which is to be patched at prepare time. The high order bit of the pointer determines whether a byte or a word pointer is being initialized. If .(0:1)=1, then the contents of the code module word specified by the word offset in .(1:15), and the correction added at prepare time, are byte offsets. If .(0:1)=0, they are word offsets. (It is believed that the high order bit of the code module word pointed to is also interpreted in this way. That is, if either high order bit is on, either in the header pointer or in the code module word, then the address is to be a byte address.)

3.5.2.4. Secondary DB initial values headers

This header may be used to place initial values into the program unit's secondary DB array. The word which follows the header descriptor word gives the offset into the secondary DB array at which the first of the given initial values is to be placed.

The third word of the header has two fields. The .(0:1) field determines whether a byte initialization or a word initialization is to be performed. If .(0:1)=1, then the second word of the header is a byte offset, and the fourth word of the header is a byte count giving the length of the initial values in the header. In this case, the initial values begin in the fifth word of the header and continue for as many bytes as the fourth word indicates. If .(0:1)=0, then the second word of the header is a word offset. In this case, the initial values begin in the fourth word, and continue to the end of the header.

The .(1:15) field of the third word gives a replication factor. The initial values specified in the header will be placed in successive locations in the secondary DB array as many times as indicated by this field. Thus, if the initial values are "xxyxx" and the replication factor is 2, then "xxyxxxxyxx" will be placed into the secondary DB array, beginning at the location specified in the second word of the header.

3.5.2.5. TRACE/3000 header

This header provides information for use at run time by TRACE/3000. After the header descriptor word, there is a word pointing to a linked list in the code module. After this, beginning in the third word of the header, and continuing to the end of the header, is data which is believed to be initial values of some sort. No further information is available as of this writing about this header type.

3.5.2.6. Global variable headers

It is possible to declare a variable GLOBAL in one program unit, EXTERNAL in another, separately compiled program unit, and have the MPE segmenter resolve all references to the variable. (SPL/3000 is the only Hewlett Packard language allowing explicit declaration of GLOBAL or EXTERNAL attributes.) Following the header descriptor word is a data descriptor word, which gives the type and structure of the variable. The fields of this data descriptor are as follows:

- .(0:4) the mode of the variable (0=null, 1=value, 2=reference)
- .(4:6) the variable's structure (0=simple variable, 1=pointer, 2=array)
- .(10:6) the type of the variable (0=null, 1=logical, 2=integer, 3=byte, 4=real, 5=double, 6=long, 7=complex, 8=label (passed SPL fashion), 9=character (as in FORTRAN/3000), 10=label (passed in in FORTRAN/3000 fashion), 11=any)

In the left byte of the third word of the header is the run time DB relative address of the variable. (Global storage address assignments for primary DB are normally made by a compiler while compiling an outer block, and are not in any way relocated by the segmenter.) The .(8:4) field of the third word is reserved and should be set to zero. .(12:4) contains the length of the name of the variable, in bytes. The name itself begins in the left byte of the fourth word, and continues for as many bytes as necessary. The name is always an integral number of words in length, and so a byte is sometimes wasted.

3.5.2.7. External variable headers

A variable declared EXTERNAL is to be matched at prepare time with a variable declared GLOBAL in some other program unit. The first word of the header is of course a header descriptor word. The second word is a data descriptor word, which has the format described in section 3.5.2.6. Following the second word is a name field. The .(0:1) bit of the first word of the name field is a 'trace' bit. If it is on, it indicates that the variable may be traced by TRACE/3000 at run time. .(1:3) is reserved, and should be set to zero.

If the trace bit is on, then in the word immediately following the name field is an offset into the TRACE/3000 symbol table. If the trace bit is off, this offset is not present.

Following the name field, and the TRACE/3000 symbol table offset, if present, is a series of pointers, each of which is an offset into the code module. Each points to the first of a list of instructions to be patched with the address of the appropriate GLOBAL variable. Each of the instructions to be patched must be a memory reference instruction, since GLOBAL variables will always reside in the primary DB area. The address fields of the instructions to be patched (the right byte in memory reference instructions) serves as the link field for the list. The links are self-relative backward pointers. Each list is terminated by a zero pointer.

There is no explicit indication of the number of pointers in the header. This must be deduced from the length of the header.

3.5.2.8. Primary DB headers

For the purposes of the MPE segmenter, primary DB words are classified into word pointers, byte pointers, and data. After the descriptor word in this header there is a series of words, each of which is divided into eight two-bit fields. All these fields, in order of occurance, correspond to primary DB locations. The first if for DB+0, the second for DB+1, and so on. The values of the fields are interpreted as follows:

0 the initial value is not an address
1 the initial value is not an address
2 the initial value is a word address which points to the secondary DB area
3 the initial value is a byte address which points to the secondary DB area

Initial values in the header that are addresses are relative to the beginning of the program unit's secondary DB area. The entry, after the array of two-bit-field words, contains initial values. There must be PDB (see section 3.4.2.2) two-bit fields, and PDB initial values.

There may be a slack word between the two-bit-field array and the initial values. Because of this, the initial values should always be accessed from the end of the header. That is, if P is an integer pointer to the last word of the header, then P(-(PDB-1)) accesses the first initial value.

Normally, only an outer block program unit would make use of this header type. Non-outer block program units should not be allocating primary DB storage, and the value of PDB for them should be zero.

3.5.2.9. Common variable headers

The MPE segmenter allocates secondary DB storage for all common blocks. In order for a program unit to access a variable in common, it must use this header. For each common variable referenced in one of these headers, the MPE segmenter will allocate a pointer in the primary DB area, and properly initialize it to point to the common variable. Specified instructions will be patched with the address of this pointer.

Following the header descriptor word is an integer which gives the length in words of the common block to which the header applies. Beginning in the third word is a name field, giving the name of the common block to which the header applies (blank common is named "COM'"). The .(0:4) field of the name field is reserved and should be set to zero.

Beginning in the word immediately following the name field is a series of variable descriptors. There is no explicit indication of the number of variable descriptors in the header. This must be deduced from the header's length and contents. Table 3.5.2.9 gives the format of variable descriptors.

It must be noted that if the trace bit (P(0).(1:1)) is not on, then the displacement into the TRACE/3000 array (P(2)) is not included. It is simply omitted, and the list heads move up to fill in its place.

Table 3.5.2.9 -- Variable Descriptor Formats

Field	Contents
P(0).(0:1)	0=DB pointer is to be of type word, 1=DB pointer is to be of type byte
P(0).(1:1)	'trace bit'; 0=variable will not be traced by TRACE/3000 at run time, 1=variable may be traced
P(0).(2:14)	the number of lists of instructions which are to be corrected (there are this many list heads later in the varible descriptor)
P(1)	the displacement within the common block of the variable
P(2)	displacement within a TRACE/3000 array of information about the variable (NOTE: this field is present only if the trace bit (P(0).(1:1)) is set; otherwise it is completely omitted)
P(3) to P(2+P.(2:14)) -or- P(2) to P(1+P.(2:14))	the list heads of the lists of instructions to be patched; each list head is an offset into the code module to the first word of a list (the lists are formed the same as the code module lists used by EXTERNAL variable headers, described in section 3.5.2.7)

3.5.2.10. FORTRAN logical units table headers

This header indicates which FORTRAN logical units are referenced by the program unit. The MPE segmenter will construct the FORTRAN logical units table from the information contained in FLUT headers. After the header descriptor word there are exactly seven words. These words contain a bit map, in which the first bit corresponds to logical unit number zero, the second to logical unit 1, the third to LU 2, and so on. If a bit is on, the corresponding logical unit will be included in the FLUT table at run time. The bits are numbered from left to

right. The 'left-most' word is the one which occurs nearest to the header descriptor word. Legal LU's range from 1 to 99, inclusive.

3.5.2.11. Format headers

Formats which include an 'H' specification, and are referenced in a READ statement, must be globally located to retain between calls to the program unit values read into the 'H' specification. This header allows that. It contains a format string which is to be placed in the secondary DB area.

After the header descriptor word is a word which gives a word offset into the code module. The code module word thus indicated is the first of a list of words to be initialized at prepare time with the DB relative address of the format string. Within the list in the code module, the .(2:14) field of each member of the list is a self-relative, backward pointer to the next list element. A link of zero terminates the list. If the .(0:1) field of such a code module word is set on, then the DB relative pointer placed into that word is to be a byte address; if .(0:1)=0, then the DB relative pointer placed into the code module word is to be a word address. The pointer placed into the word will point at run time to the beginning of the format string.

The third word of the header gives the length, in bytes, of the format string. The fourth and following words, as many as necessary, contain the format string itself.

:EOF: