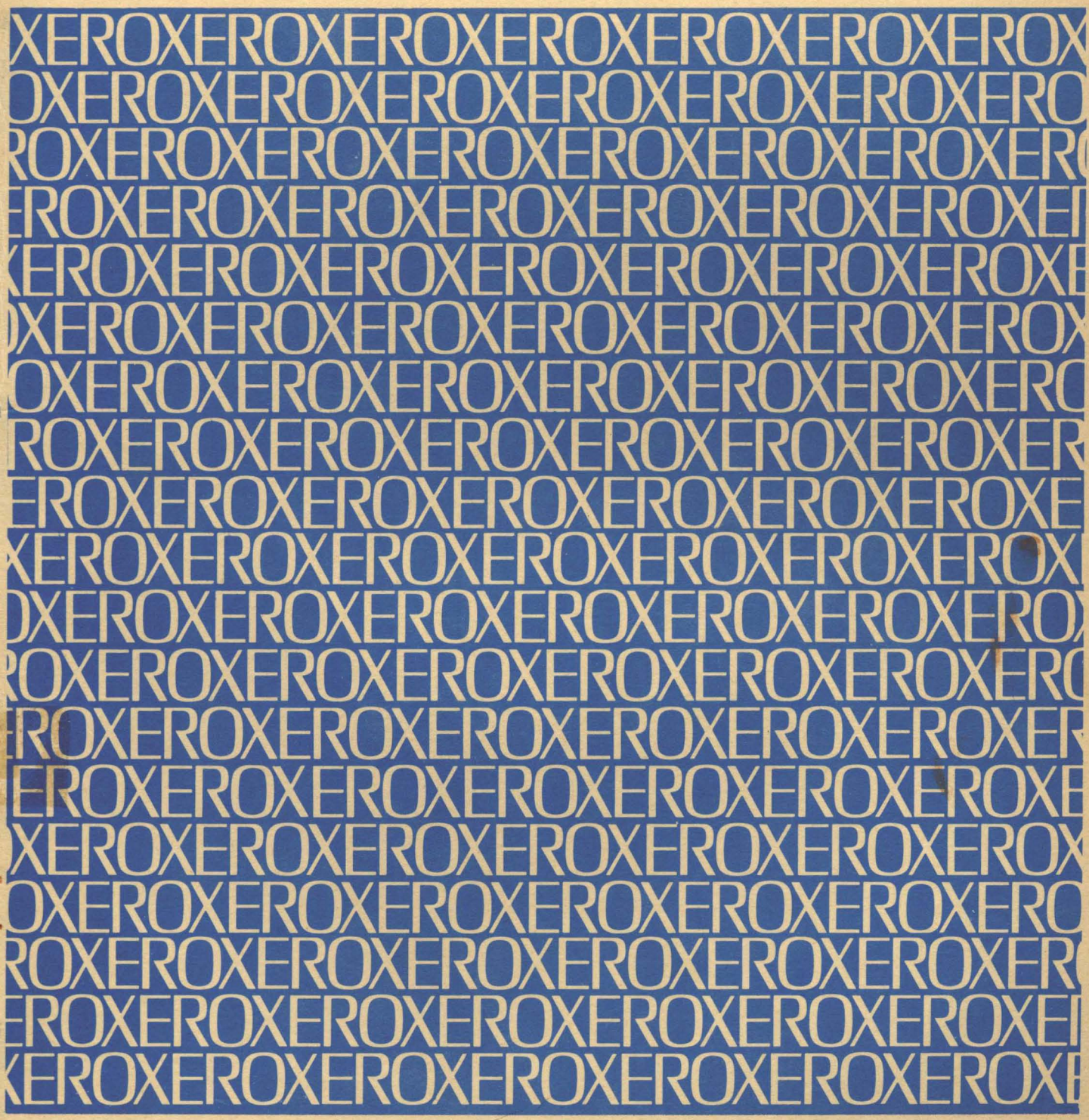
D 8

Xerox BTM/BPM/UTS

Sigma 5-9 Computers

Overlay Loader
Technical Manual



90 18 03B

701 South Aviation Boulevard
El Segundo, California 90245
213 679-4511

XEROX

Xerox BTM/BPM/UTS

Sigma 5-9 Computers

Overlay Loader

Technical Manual

90 18 03B

June 1972

Price: $5.00

Printed in U.S.A.

REVISION

The Overlay Loader described in this manual operates under the B00 version of UTS and the F01 version of BPM/BTM. Changes in the text marked by revision bars apply only to UTS. The previous (A) edition of this manual completely describes the loader operating under the F01 version of BPM/BTM.

RELATED PUBLICATIONS

Title	Publication No.
Xerox Sigma 5 Computer/Reference Manual	90 09 59
Xerox Sigma 6 Computer/Reference Manual	90 17 13
Xerox Sigma 7 Computer/Reference Manual	90 09 50
Xerox Sigma 8 Computer/Reference Manual	90 17 49
Xerox Sigma 9 Computer/Reference Manual	90 17 33
Xerox Universal Time-Sharing System (UTS)/TS Reference Manual	90 09 07
Xerox Universal Time-Sharing System (UTS)/SM Reference Manual	90 16 74
Xerox Universal Time-Sharing System (UTS)/OPS Reference Manual	90 16 75
Xerox Batch Processing Monitor (BPM)/System Technical Manual	90 15 28
Xerox Batch Time-Sharing Monitor (BTM)/Edit Subsystem Technical Manual	90 19 11
Xerox Batch Time-Sharing Monitor (BTM)/Delta Subsystem Technical Manual	90 18 79
Xerox BPM/BTM/UTS/System Generation Technical Manual	90 18 77
Xerox Batch Processing Monitor (BPM)/BP, RT Reference Manual	90 09 54
Xerox Batch Processing Monitor (BPM) and Batch Time-Sharing Monitor (BTM)/OPS Reference Manual	90 11 98
Xerox Batch Time-Sharing Monitor (BTM)/TS Reference Manual	90 15 77

Manual Content Codes: BP – batch processing, LN – language, OPS – operations, RBP – remote batch processing, RT – real-time, SM – system management, TS – time-sharing, UT – utilities.

CONTENTS

FIGURES

PREFACE

This document describes the purpose and architecture of the Overlay Loader within the environment of BPM or UTS. It is assumed that the reader is familiar with the usage of BPM/UTS Monitor services as well as the Sigma Standard Object Language (see the BPM/BTM/SM Reference Manual, 90 17 41).

GLOSSARY

CCI (Control Command Interpreter): a processor (brought into core by the Monitor) which reads the !LOAD card and records the information in an LOCCT Table.

core image: that part of a load module which is laid into core at execution time.

core library: for UTS, a special collection of files under the :SYS account for association with FORTRAN programs.

DCB (Device Control Block): a table for use by the Monitor in performing an I/O operation.

DCB Name Table: a loader-built table which directs the Monitor to the location of a particular DCB within a program.

declaration stack: a Loader stack which serves to keep track of the declarations made in a given ROM.

DEFCOM: a processor which outputs a special type of load module.

expression stack: for any segment, a collection of expressions defining DEFs and forward references and expressions whose values are to be placed in the segment's core image.

extended memory mode: a mode in which the Loader builds core images and relocation dictionaries in page-sized records within a file on the RAD.

HEAD: a key to one of the records of a load module file, the record containing basic size and source information.

idB: a CCI-built table containing information from the BI device (when BI is specified on the LOAD card).

idD: a file built by CCI on the basis of !MODIFY cards following the !LOAD card.

idG: the file name the Loader uses to access information specified by the GO option.

idL: the file name assigned to a load module if no name is specified via the LMN option.

idX: the name of the intermediate file used during the extended memory mode to build standard (i.e., nonpaged) core image and relocation dictionary records (BPM only).

JIT (Job Information Table): a Monitor table of information pertinent to the job currently in execution.

library: the term ascribed to two files, :LIB and :DIC, which are constructed by the Loader.

load item: a string of bytes representing a "clause" in object language.

load module: a keyed file which is output by the Loader (and several other processors).

LOCCT (Load Control Command Table): a table which the Loader must access for its own control card input.

object language: the language generated by assemblers and compilers to convey information to the Loader.

PASS3: a processor which calls the Loader to form a load module.

path: a collection of segments of a program which can reside in core at the same time.

REF/DEF Stack: a Loader-built stack for each segment whose entries contain values for control sections, external names (DEFs, REFs, SREFs), and forward references.

relocation dictionary: a record constructed by the Loader which indicates how to relocate each word of a corresponding core image record.

ROM (Relocatable Object Module): a type of input component to the Loader which was generated by an assembler or compiler.

segment: a piece of a program which may be replaced in core by another piece of the program.

stack path: the collection of REF/DEF or expression stacks belonging to the segments on a given path.

system id: a job-oriented identification number determined by the Monitor and supplied to the Loader via the LOCCT Table.

TCB (Task Control Block): a Loader-built table containing the user's temp stack and areas for system use.

TREE: a collection of tables reflecting the overlay structure of a program.

1.0 ENVIRONMENT

1.1 INTRODUCTION

The purpose of any loader is to translate and unite its input (ROMs and libraries) into such a form that the output (a load module) may be executed under the target operating system. Accordingly, the Overlay Loader performs those functions which might be expected of any loader operating under BPM or UTS:

a. Process ROMs producing continuous sections of data, procedure, and DCBs (or static data if BPM), insuring a page boundary for the three protection types (00, 01, 10, respectively).

b. Satisfy REFs among the ROMs.

c. Access "libraries" to satisfy PREFs.

d. Build DCBs.

e. Build a DCB Name Table for Monitor use.

f. Build a TCB.

The special characteristics of the Overlay Loader are identified as follows:

a. Create Overlay Programs

An overlay program is one which has only one piece (segment) resident in core permanently. The other segments are called by the M:SEGLD procedure and brought into core as needed. These segments may reside (at different times) in the same core area, thus reducing the amount of core required to house the entire program.

Since, in general, a program may consist of three areas (one per protection type), each beginning on a page boundary, the Overlay Loader must have the ability to create the three trees, each beginning on a page boundary.

b. Reference Loading

If the user does not choose to maintain responsibility for calling the segments of an overlay program (by explicitly using M:SEGLD), he may direct the Loader to insert the M:SEGLD code into his program by specifying REF or BREF on the !LOAD card. This code is built, in the BREF mode, wherever there is a branch type instruction involving a REF to a higher segment. In the REF mode, it is built wherever there is any expression whatsoever involving a REF to a higher segment.

c. Load Module Libraries

It is desirable to maintain libraries of frequently used routines which are themselves already in load module form, since subsequent inclusion of a library module would be faster than processing the original ROM language.

d. Relocatable Load Modules

The Loader creates a relocation dictionary which allows subsequent placement of the load module into a core area other than the one at which it was originally biased. Relocation is accomplished via a BIAS option on the !RUN command for BPM. (NOTE: UTS does not allow a BIAS on the !RUN card; hence for UTS, the only use of the relocation dictionary is in the case of merging a library load module into another program.)

e. Dummy Sections

The Loader has the ability to recognize dummy sections of the same name in various modules and to allocate on the basis of the largest one encountered. This feature is generally used in large FORTRAN programs which rely heavily on COMMON (COMMON is a form of dummy sectioning).

1.2 SYSTEM INTERFACE AND GENERAL OPERATING CHARACTERISTICS

1.2.1 Loader Operation Under BPM or UTS

The Loader operates under BPM or UTS and produces either BPM or UTS load modules. These load modules are not interchangeable due to differences in the format of the HEAD record and in the allocation of the DCB area (see Section 3.2.1). Neither are the Loaders themselves interchangeable. That is, the UTS Loader will not operate under BPM and vice versa, due to differences in obtaining memory (see Section 2.4). An assembly parameter will select those areas of loader code which are unique to BPM or UTS. The parameter is MODE. At assembly time, in each source module except the last, this parameter must be set to 0 for BPM and 1 for UTS.

1.2.2 Loader Entry/Exit

Loader entry/exit is via CCI or the SYSGEN PASS3 processor (as a result of !LOAD or !PASS3). If the Loader is entered via CCI, the !LOAD and !TREE cards and (optionally) the BI device are read by CCI and packaged into tables (see Tables, Section 3.1.1) prior to entry. If entry is through PASS3, these tables are accessed from a previously existing file (created by the !LOCCT processor) and presented to the Loader in the same form that CCI would have presented them. The Loader decides which return to execute (an M:EXIT to CCI or an M:LDTRC to PASS3) on the basis of register or JIT input. Also, for UTS, if the Loader is to exit to PASS3, it must first "release" all memory which it obtained (via M:GP or M:GCP or M:GVP).

1.2.3 What the System Does With the Loader's Output

A !RUN command will cause the "Program Loader" (in BPM-PRGMLDR, in UTS-FETCH) to access the load module file, modify and/or relocate it, lay it into core per the dictates of its HEAD and TREE records and transfer control to the START address (whereupon the program is "in execution"). Figure 1 shows the user program as it sits in core during execution.

If the program is overlaid, at some point it will issue an M:SEGLD call. (This was part of the user's code or was inserted by the Loader per the REF/BREF option.) Since a copy of the TREE is always a permanent part of the root segment (protection type 01), the segment loader has all the information it needs to access the desired segment, deposit it in its destination, and record the fact that the segment is now in core (to avoid unnecessary reloading in the future).

Branching between segments is the user's responsibility if he issued an explicit M:SEGLD. If REF or BREF is in effect, the branching is automated by the Loader-built table entry.

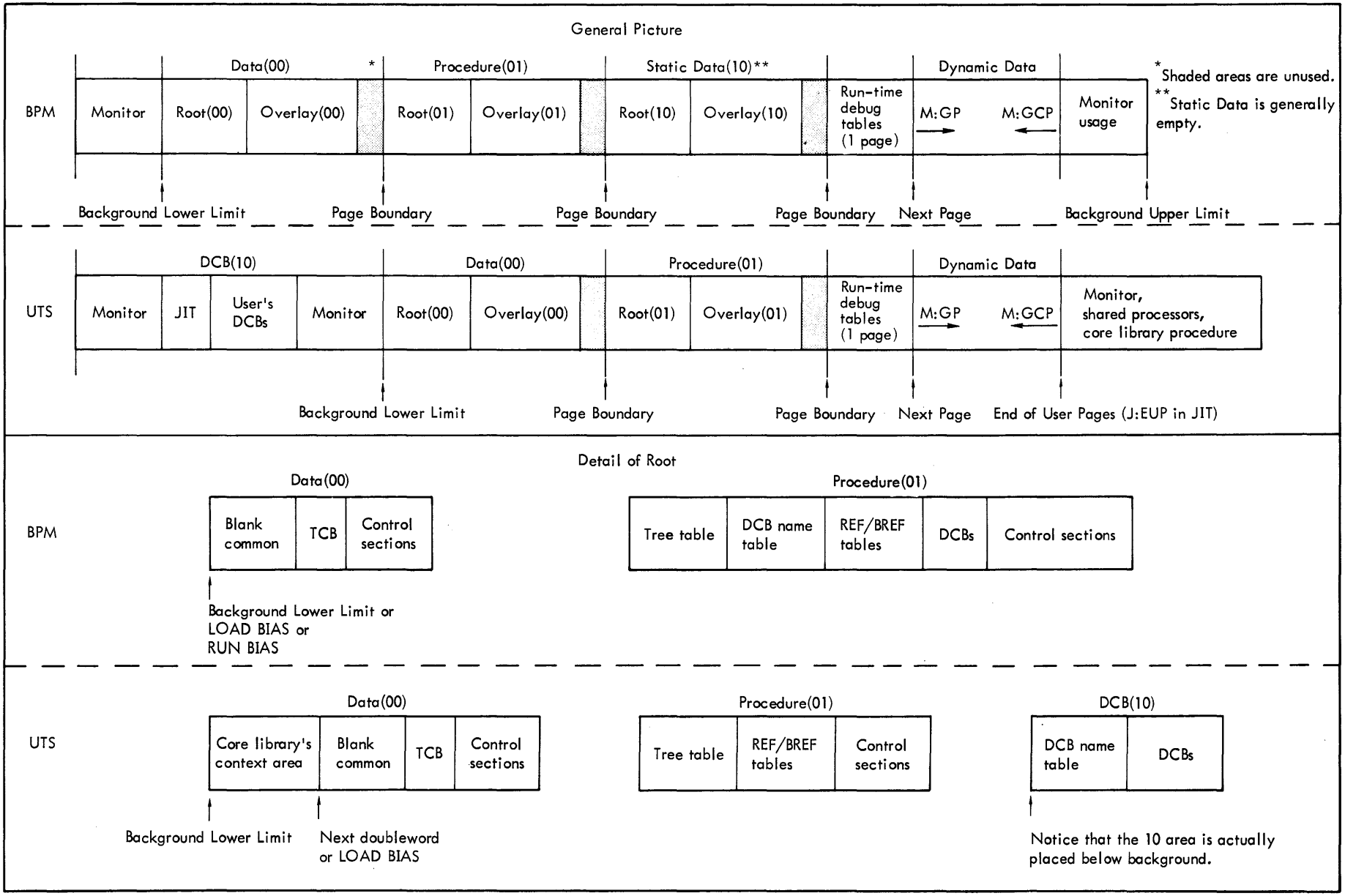


Figure 1. Load Module Layout at Run-Time

2.0 GENERAL OPERATING CHARACTERISTICS

2.1 FUNCTIONAL OVERVIEW

2.1.1 Loader Terminology

At this point, it might be well to review some fundamentals of the object language and define some terminology relative to the Loader.

a. Declaration Numbers

Within a given ROM, all control sections, DEFs, PREFs, SREFs are declared; that is to say, each is assigned a "declaration number". (The ROM assigns declaration numbers consecutively.) In an expression which involves any of these items, the ROM refers to them via their declaration number. The Loader, therefore, must remember these numbers; it does so by building a declaration stack as the numbers are encountered within the module. (The stack is destroyed at module end since it has no meaning for the next module.) An entry in the declaration stack is simply a pointer to the proper entry in the segment's REF/DEF stack (which will eventually contain the complete story about that declaration).

b. Dummy Sections

Within a ROM, a dummy control section is treated as both a DEF and a control section. In particular, the ROM must first declare the dummy section's name (the label that is to be associated with the first location of the section) as an ordinary external definition. Subsequently, the ROM declares the dummy section itself as a control section (via 'Declare Dummy Section'). This declaration refers to the previously declared label, thereby associating the name with the dummy control section.

c. Expressions

The value of a DEF, Origin, Start or forward reference is given to the Loader from the ROM via an expression. Load items to be placed in the core image also involve expressions. An expression consists of operators (control bytes) which operate on constants, declarations, and forward reference numbers. Thus, an expression might say "add the value of declaration 5 with the constant X'10' ". When the Loader wants to calculate the result of ("evaluate") this expression, it first looks in the fifth entry of the declaration stack to get a pointer to the proper REF/DEF stack entry. Next it adds the value word of that REF/DEF entry to the expression accumulator and then adds the value X'10' to the expression accumulator.

d. Forward Reference Numbers

Within a ROM we will encounter expressions involving forward references. These are referred to via random numbers. Therefore, the Loader must keep track of them in a similar way that it keeps track of declaration numbers. This is done by creating an entry in the REF/DEF stack containing the reference number. When a forward reference number is encountered in an expression, the Loader searches the REF/DEF stack for a match. If none is found, a new entry is created. Since the numbers are meaningless for the next module, the Loader "releases" them at module end.

Forward references are of two types: those which can be resolved by module end or sooner, and those which cannot be so resolved. The latter type consists of forward references whose defining expressions contain REFs or DSECTs. When the expression to define the forward reference is encountered, it will indicate which of the above was meant (define forward reference (DFREF) or define forward reference and hold (DFREFH)). A DFREF expression implies that the corresponding forward number is now closed and invalid;

a new expression involving that number refers to a new forward reference. The Loader must mark its REF/DEF entry as such ("release" it). On the other hand, a DFREFH expression implies that the number may occur within another expression. Therefore, it is still valid and cannot be released until module end. Notice that the number is always released at module end, even though the forward reference itself may not be resolved yet.

e. Files, Segments, and Paths

A segment is made up of files (ROMs or load modules) and is a piece of the target load module. A segment may be overlaid by another segment. A path of an overlay structure is a set of segments which reside in core at the same time. The root is the segment which is always in core. In Figure 2, there are three paths: S0-S1-S3, S0-S1-S4, and S0-S2. Given a segment we may speak of its back-link, forward-link, and overlay-link. (The forward-link is also called the "sublink".)

Referring again to Figure 2:

Segment	Back-Link	Forward-Link	Overlay-Link
S0	None	S1	None
S1	S0	S3	S2
S2	S0	None	None
S3	S1	None	S4
S4	S1	None	None

f. Loader Stacks

The Loader forms three stacks: the declaration stack, REF/DEF stack, and expression stack. The declaration stack is created and destroyed for each ROM. An entry in the declaration stack is simply a pointer to that entry in the REF/DEF stack which describes the declaration. The REF/DEF stack is really a misnomer since it includes an entry for every declaration (control section, DEF, REF, SREF) as well as for forward references.

The expression stack contains defining expressions for DEFs and forward references, as well as expressions whose value is to be added to a word in the core image itself (core expressions).

The components of an expression are operators (control bytes) acting on declaration numbers, forward reference numbers, and constants. The value (result of performing the operations, e.g., add value of declaration, add constant, etc.) is either placed in the VALUE word of the REF/DEF stack or in the core image if it is a core expression.

The Loader creates a REF/DEF and expression stack for each segment. These stacks are created along a path. The Loader's stack area will develop in the same way that the segments are overlaid. In Figure 2, if we are working on S4, then the stacks for S0 and S1 and S4 are in core. If stack S4 is in core, stack S3 will not be, since they are on different paths. (This implies, incidentally, that if one segment is to communicate with another via REFs and DEFs, they must lie on the same path.) We refer to a "stack path" as the set of stacks belonging to a given path.

2.1.2 The First Pass

The first pass gathers all information relative to the sizes of major pieces of the load module (i.e., the size of each protection type per segment and the stack sizes). Additionally, the first pass provides the second pass with an efficient means of developing the core image by constructing the REF/DEF and expression stacks. As it scans each input component (ROM or load module),

Pass One examines only that information necessary to accomplish these two functions; viz. size computation and stack construction. Any information not relevant to these functions is ignored until Pass Two.

If the component is a ROM, the sizes are to be found in the "declare control section" load items. Pass One accumulates each control section size in the appropriate protection type of the TREE. A REF/DEF entry is also built for each "declare control section" load item, as well as for load items which declare names as forward references. (For each name declaration, either a new entry is added to the REF/DEF stack or an old one is modified.) Expression stack entries result from load items which define external DEFs or forward references. (When a new entry is made in either the REF/DEF or expression stack, the size of that stack is updated in the TREE.) All load items dealing with the content of the core image (e.g., "load relocatable") are ignored.

If the component is a load module, the HEAD and TREE records contain the sizes of the core image and the stacks. These are added to the TREE. The load module's stacks are then merged with the ones being constructed. The core image and relocation dictionary records are ignored until Pass Two.

At the end of processing each segment's explicit element files, the REF/DEF stack is scanned for all PREFs except those having names starting with M: or F: (for which the Loader will build DCBs). For each PREF found, Pass One searches those libraries specified on the !LOAD card for a load module which will satisfy the PREF. If it finds one, the load module's name is added to the list of input files (ROM Tables), the size is recorded in the TREE and its stacks are merged with the ones being built.

The sequence of processing the overlay structure is from the root segment outward, as shown in Figure 2. Figure 3 shows the general flow during Pass One.

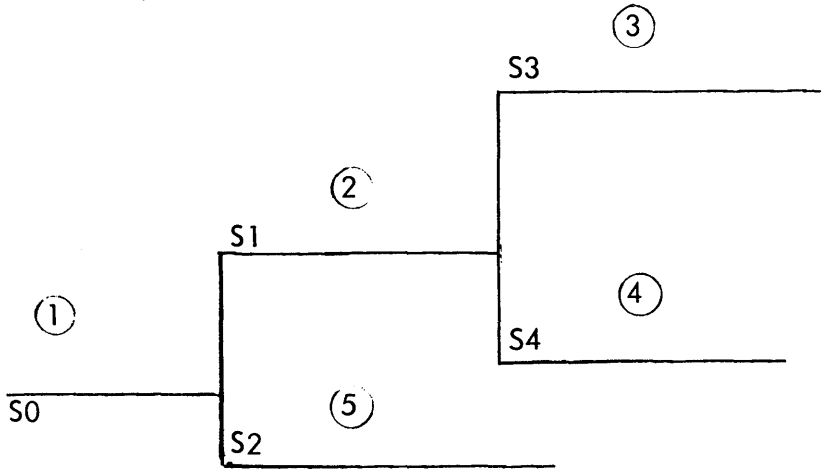


Figure 2. Segment Processing Sequence, Pass One

At the end of the first pass we have:

a. sizes of the segments per protection type, including the sizes of modules obtained from the library. These sizes are in the Tree Tables.

b. a REF/DEF and expression stack for each segment written to the RAD. The stacks are structurally complete. They include the merged library stacks. The value and resolution for each REF/DEF entry is not determined until the second pass.

c. defining expressions for DEFs and FREFs in the appropriate expression stacks. The expression stacks also include expression stacks from library load modules.

d. sizes for the REF/DEF and expression stacks in the Tree Tables.

e. ROM Tables augmented by the names of library load modules pulled in as a result of satisfying PREFs.

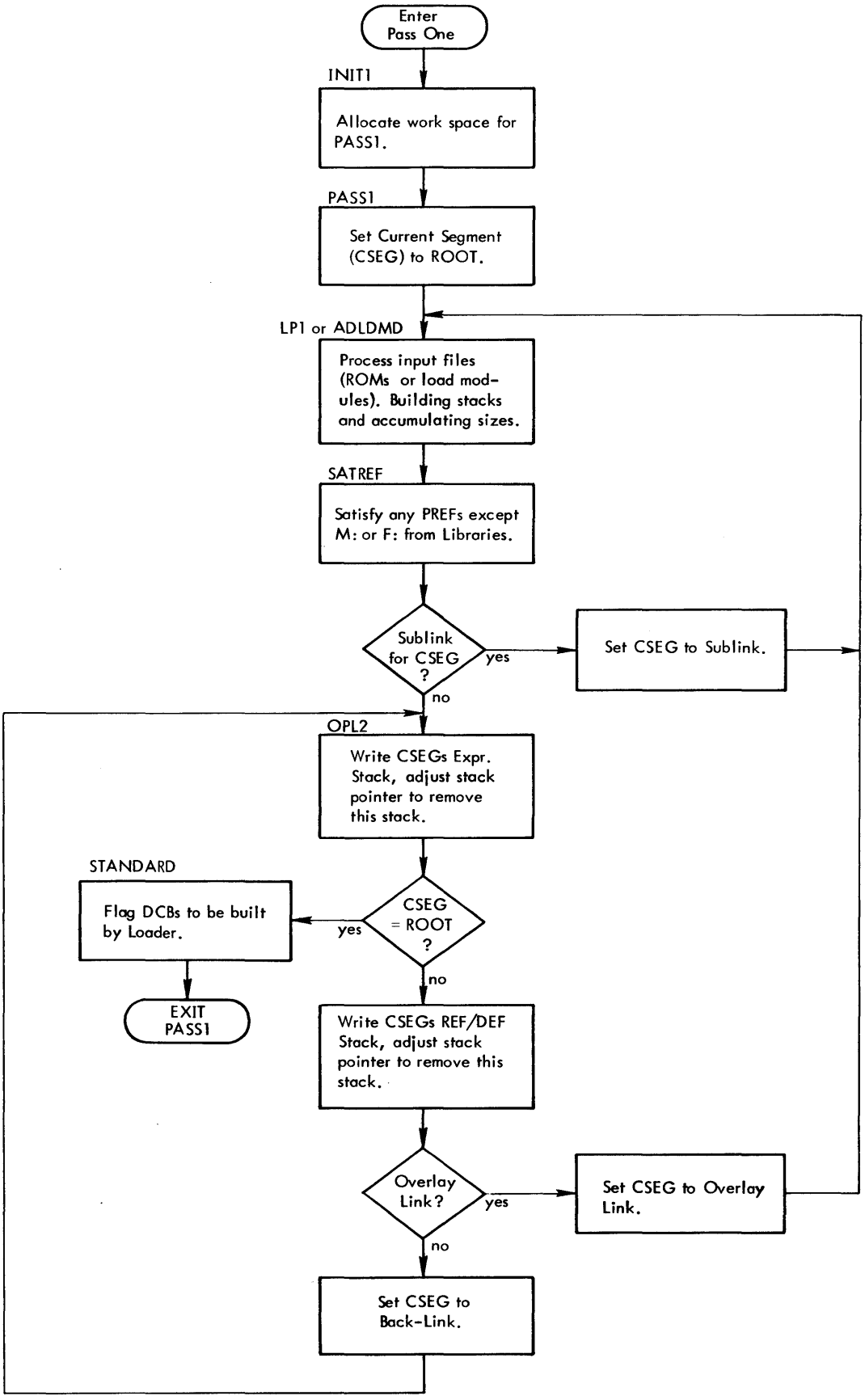


Figure 3. The First Pass – General Flow

2.1.3 The Second Pass

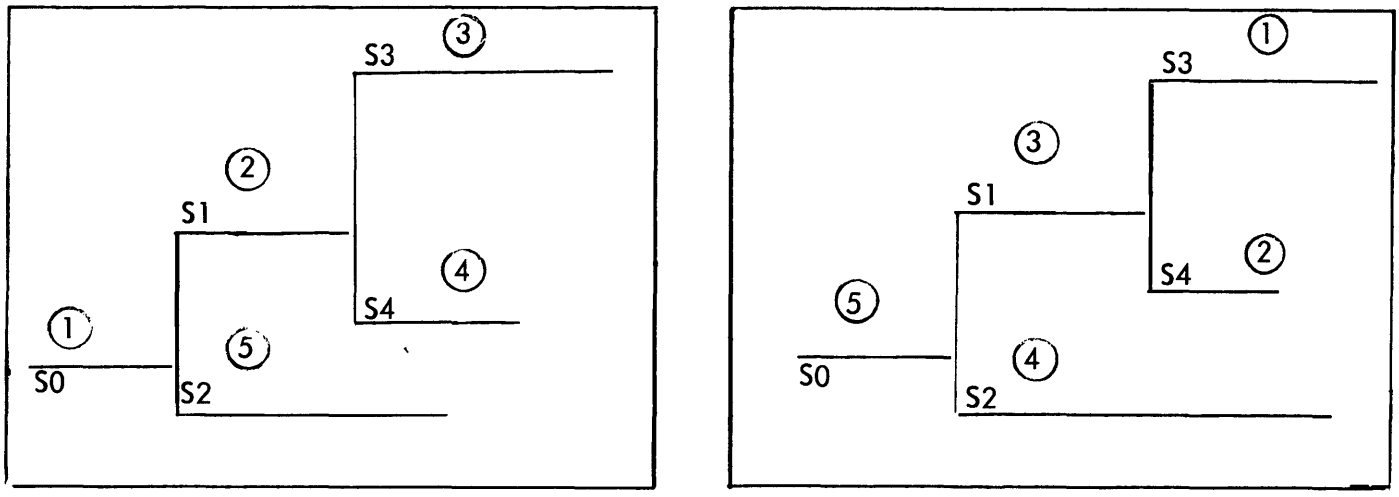
This pass develops the actual core images and relocation dictionaries and writes the load module to the RAD.

Based on the sizes known from PASS1, core is partitioned into the stack area and buffers for the core images and relocation dictionaries. If this partitioning is not possible, Pass Two goes into "extended memory mode", meaning that the core images and dictionaries will be developed within an intermediate RAD file, page by page.

The expression and REF/DEF stacks for an entire path are brought into core and, from the size and protection type of every control section, locations are assigned to all of the sections (the control sections are "allocated") along this path.

After evaluating and defining all possible DEFs and FREFs, Pass Two is now in a position to reread the input files (proceeding backwards along a path). As it reads, it places data in the core and relocation buffers (or into the extended memory mode file, as the case may be) as per the dictates of the load items. When a segment is complete, its REF/DEF stack, expression stack, core images, and relocation dictionaries are written out (unless we are in extended memory mode, in which case processing of the paged core image and relocation dictionary records is deferred until the root segment has been constructed).

The sequence of forming the core images for an overlay structure proceeds from the sublinks back toward the root, but, as mentioned, allocation occurs forward along a path (see Figure 4).



Sequence of Core Image Allocation Per Segment

Sequence of Forming Core Image Per Segment

Figure 4. Segment Processing Sequence, Pass Two

Special attention comes into play when we reach the root segment at the end of the second pass. If extended memory mode is in effect, the load module must be reconstructed from the page records of the extended memory mode file which were created during the formation of the core image In any case, the TCB and DCBs are built, the HEAD and TREE records are written, necessary modifications per !MODIFY cards are made, the severity level is printed, and the Loader returns control to CCI or PASS3.

Figure 5 shows the general flow of Pass Two.

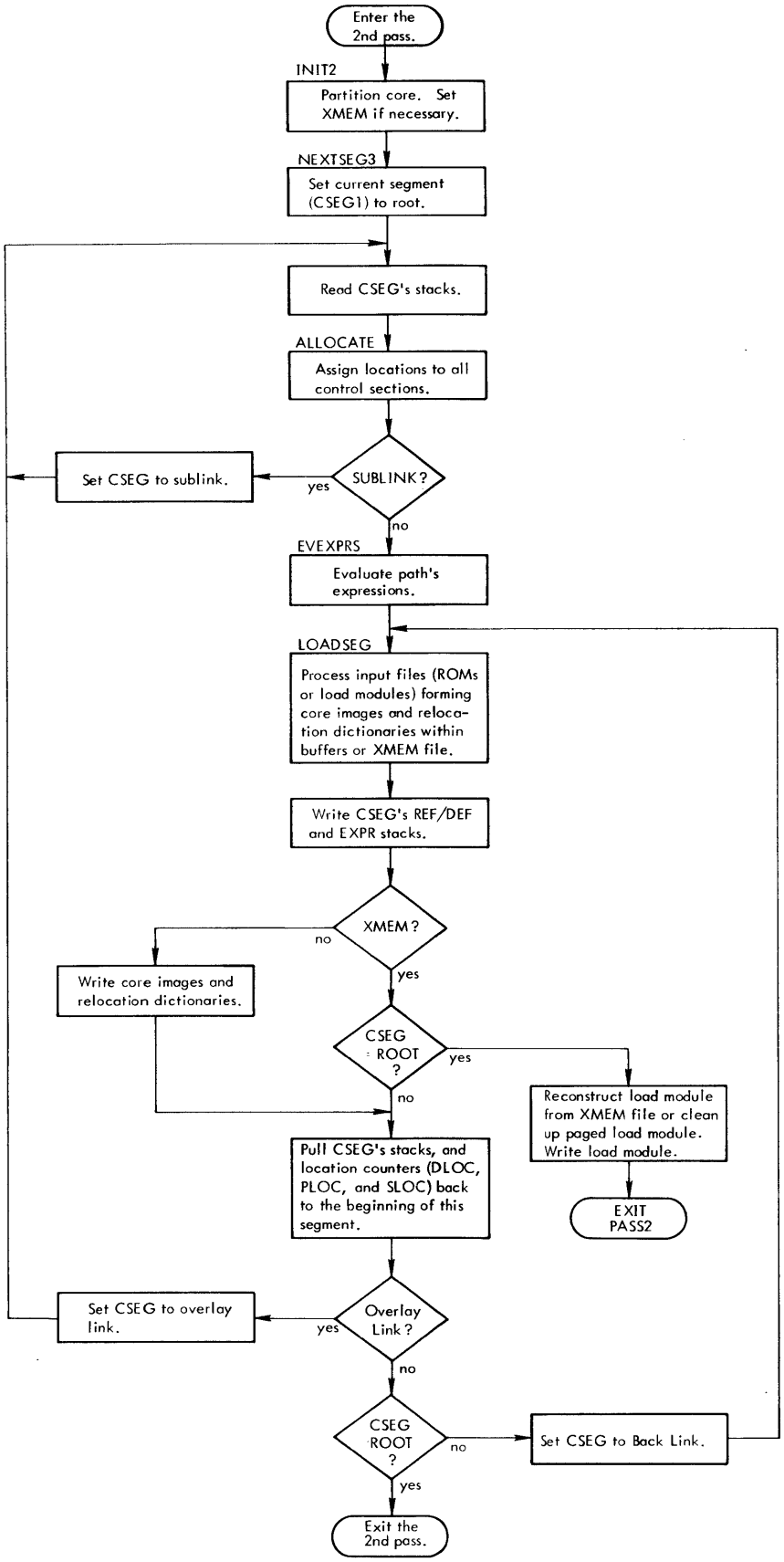


Figure 5. The Second Pass – General Flow

2.1.4 Advantages of a Two-Pass Loader

a. The primary advantage is accorded to FORTRAN programs which use Blank COMMON. A two-pass Loader has the ability to discover the largest dummy section of the same name (dummy sections are intrinsically externally defined) and to allocate accordingly. This would be, if not impossible, an extremely difficult matter for a one-pass Loader.

b. A one-pass Loader has difficulty with overlaid load modules having more than one protection type. The problem arises in determining how many pages of memory should be allocated for the 00 protection type before allocating for the 01 protection type. A two-pass Loader can compute, in its first pass, the size each protection type requires and can allocate memory accordingly for the second pass.

2.2 STRUCTURE: THE MAJOR PIECES

The Overlay Loader is a two-pass Loader; that is, the ROMs and load modules from which the target load module is constructed are read two distinct times. The Loader is composed of nine ROMs. These ROMs may be grouped according to their usage in the first or second pass.

When Used	File Name	Entry Points	Catalog No.
Throughout both passes	LDR	LOADER	704724
First Pass	IN1	INIT1	704725
	PS1	PASS1	704726
Second Pass	IN2	INIT2	704727
	PS2	PASS2	704728
	ALL	ALLOCATE	704729
	EVL	LOADSEG, EVEXPRS	704730
	WRT	WRITESEG	704731
	FIN	FINISH	706258
	MOD	MODIFY	705396

2.2.1 LDR

This ROM is a collection of frequently used subroutines, temp space, variable data, DCBs and a driver which contains the start address (LOADER) and which subsequently BALs to the first and second passes and exits.

The LDR module contains the Loader's only DATA area (one page). This area is composed of two parts: a temp stack (pointer in R0) and a collection of variable data (stack pointer, doubleword buffer pointers, location counters, etc.). See Appendix B for a description of the use of the variable data cells in the loader's STUFF stack.

Figure 6 illustrates the format of this area.

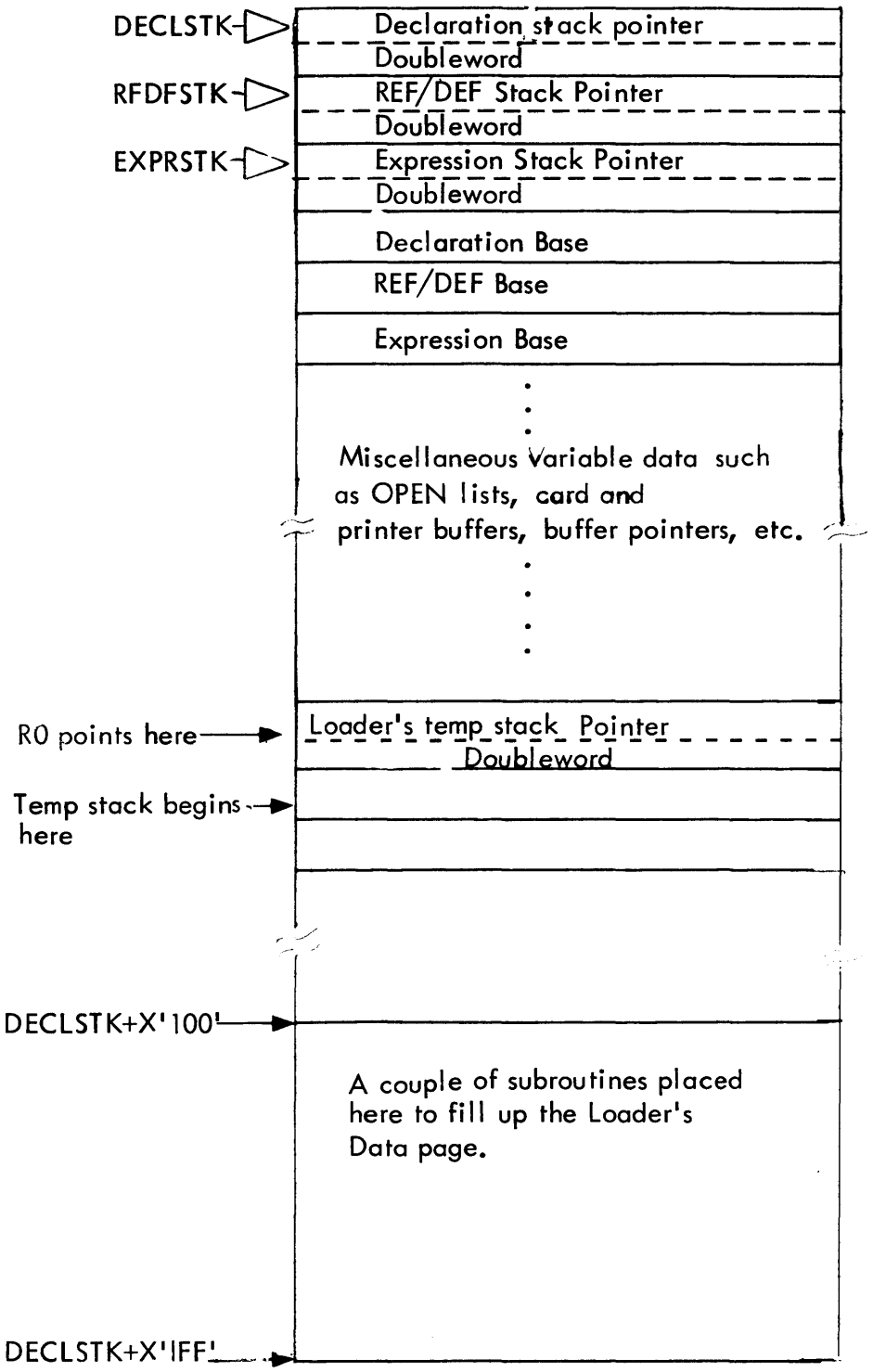


Figure 6. Loader's DATA (00) Area (Within LDR)

2.2.2 The First Pass

IN1
- entry/exit from LDR.
- allocates the work space for PS1.
- reads the LOCCT, ROM, TREE Tables.
- reads and processes the ASSIGN record.

PS1
- entry/exit from LDR.
- reads and processes ROMs and load modules, collecting the information necessary to ascertain sizes of control sections and maximum stacks.
- satisfies PREFs from libraries.
- writes out interim stacks.

2.2.3 The Second Pass

IN2
- entry/exit from LDR.
- allocates the work space for the second pass, determining if extended memory mode is necessary.

PS2
- entry/exit from LDR.
- a driver for the second pass.
- calls ALL, EVL, and WRT.
- reads current segment's stacks.

ALL
- entry/exit from PS2.
- assigns locations to all control sections.
- prints load module allocation summary.

EVL
- has two entry points, EVEXPRS and LOADSEG, both from PS2.
- evaluates expressions from PASS1 and core expressions from load modules.
- forms core image and relocation dictionary going through extended memory mode logic.
- builds reference loading table.

WRT
- entry/exit from PS2.
- creates TCB and DCBs, and the DCB Name Table.
- concatenates the pages of the extended memory mode file for a standard load module.
- cleans up the paged core image records for a paged load module.
- writes the load module to the file.

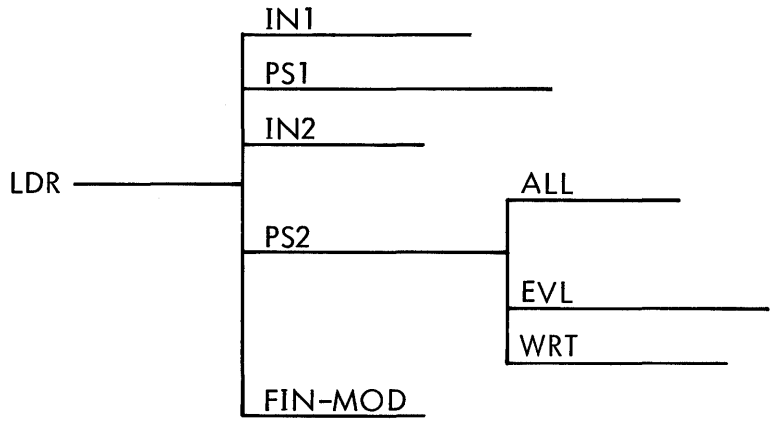
FIN
- entry/exit from LDR.
- updates and prints severity level.
- reads idD and calls MOD.
- generates load map.

MOD
- entry/exit from FIN.
- performs the modifications per !MODIFY cards which followed the !LOAD.

2.2.4 Forming the Loader

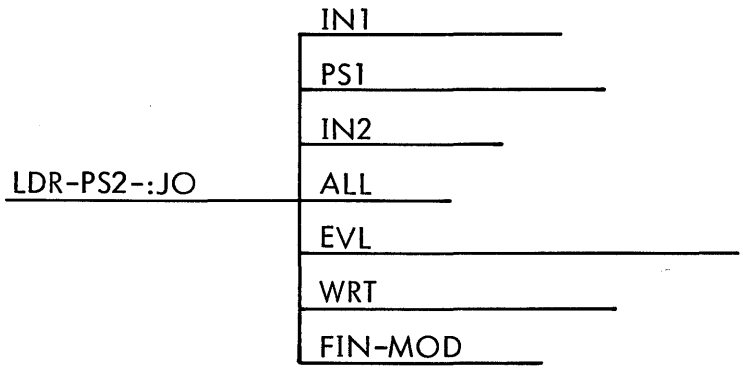
If the Loader is overlaid, it bears the following TREE structure:

```
!TREE LDR-(IN1, PS1, IN2, PS2-(ALL, EVL, WRT), FIN-MOD)
```



The UTS Loader is overlaid according to the following TREE structure:

```
!TREE LDR-PS2-:JO-(IN1, PS1, IN2, ALL, EVL, WRT, FIN-MOD)
```



The tree structure is such for the UTS Loader because, as a shared processor, the Loader is allowed only one level of overlay. Note also that the file :JO (in :SYS) must be listed as the last element file when forming the UTS version:

```
!LOAD (LMN, LOADER), (NOTCB), (NOSYSLIB), (SL, F), ;
!    (EF, (LDR), (IN1), (PS1), (IN2), (PS2), (ALL), (EVL), (WRT), (FIN), (MOD), (JO, :SYS))
```

2.3 HOW THE LOADER USES MEMORY

2.3.1 Partitioning Core for the First Pass

Recall that the first pass, after it has read the LOCCT, ROM, and TREE Tables, constructs the REF/DEF and expression stacks. (The declaration stack is volatile for each ROM.) Accordingly the partition concerns itself with only these areas. (Partitioning for the first pass is done by IN1.)

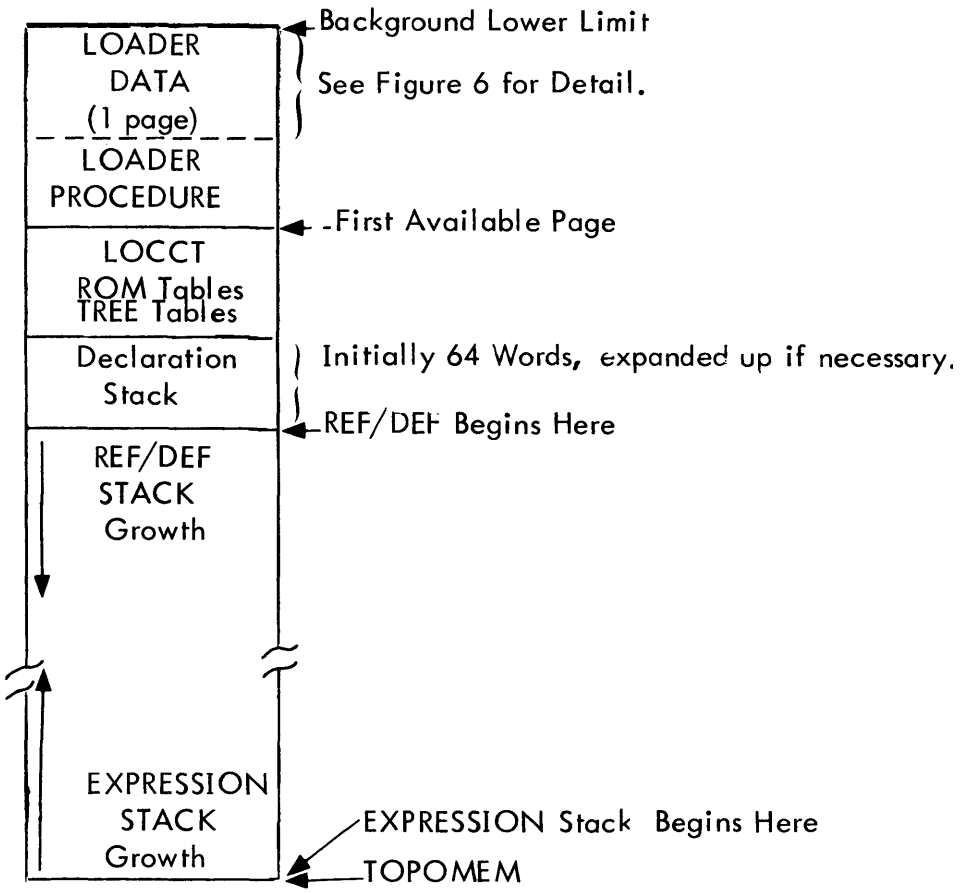


Figure 7. How the Loader Uses Memory: Pass One

If the REF/DEF and EXPRESSION Stacks meet during Pass One, processing is discontinued (JOB aborts).

2.3.2 Partitioning Core for the Second Pass

The second pass is concerned with developing the core images and relocation dictionaries (unless ABS was specified, in which case there are no relocation dictionaries). Buffers are needed to house these.

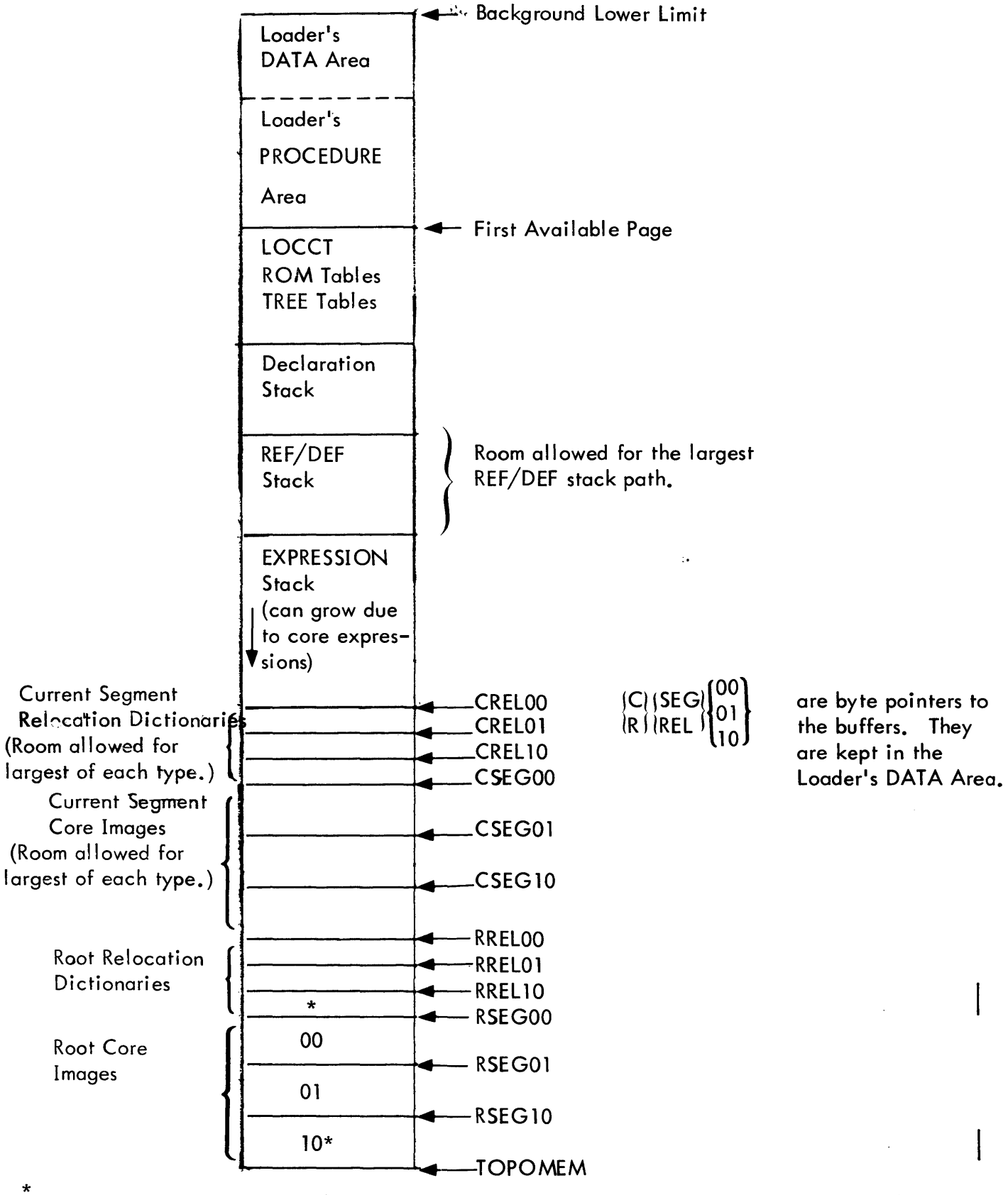
There must also be room to hold the REF/DEF and EXPRESSION stacks for the largest path. The size of the REF/DEF stack is known from PASS1, but the expression stack can grow (due to unevaluatable core expressions). Maximum declaration stack size was also retained in PASS1.

There are two partitioning schemes: nonextended memory mode and extended memory mode. IN2 (which performs the partitioning) will select the former, if space permits.

a. Nonextended Memory Mode (Fig. 8)

Two buffers are reserved for each protection type; one for the core image and one for the relocation dictionary. Such buffers are reserved for the root and for the current segment. Hence, in a full-blown relocatable TREE, there would be 12 buffers. (The reason for the double buffers is to permit a higher segment with load items in a DSECT belonging to the root to store those items into the root.)

Since the expression stack can grow, the buffer allocation begins from TOPOMEM down. (Notice that the expression stack is growing in the opposite direction than it did in the first pass.)



*For UTS, these buffers can grow due to rounding to prevent DCBs from overlapping page boundaries. If this occurs, all buffer pointers are shifted down accordingly. See Section 5.3.

Figure 8. How the Loader Uses Memory: Pass Two – Nonextended Memory Mode

b. Extended Memory Mode (Figures 9a and 9b)

If the above partition is not possible, IN2 enters extended memory mode which consists of replacing the 12 buffers with page buffers (one if ABS is specified, two if not) at TOPOMEM down. All segments, including the root, are built within these buffers. The EVL module uses these buffers to construct page records of the core images and relocation dictionaries. The file used to develop these records is either a temporary (idX) file (for a standard load module) or the load module file itself (for a paged load module). Only one page buffer is required in the latter case since a paged load module is forced ABS. Figure 9a illustrates the use of memory during the construction of the page records.

If a standard load module is to be constructed in extended memory, memory is partitioned differently at the end of the second pass (in the module WRT). Six buffers (or three if ABS) are used to concatenate the page records, one segment at a time. Figure 9b illustrates memory usage during the concatenation (sometimes called "put-together" phase) of extended memory mode.

The above partition is not required for the paged load module; instead, room is needed only for those core image records belonging to the root which are to contain loader-built tables. These records are read in successive order above DECLBAS according to protection type.

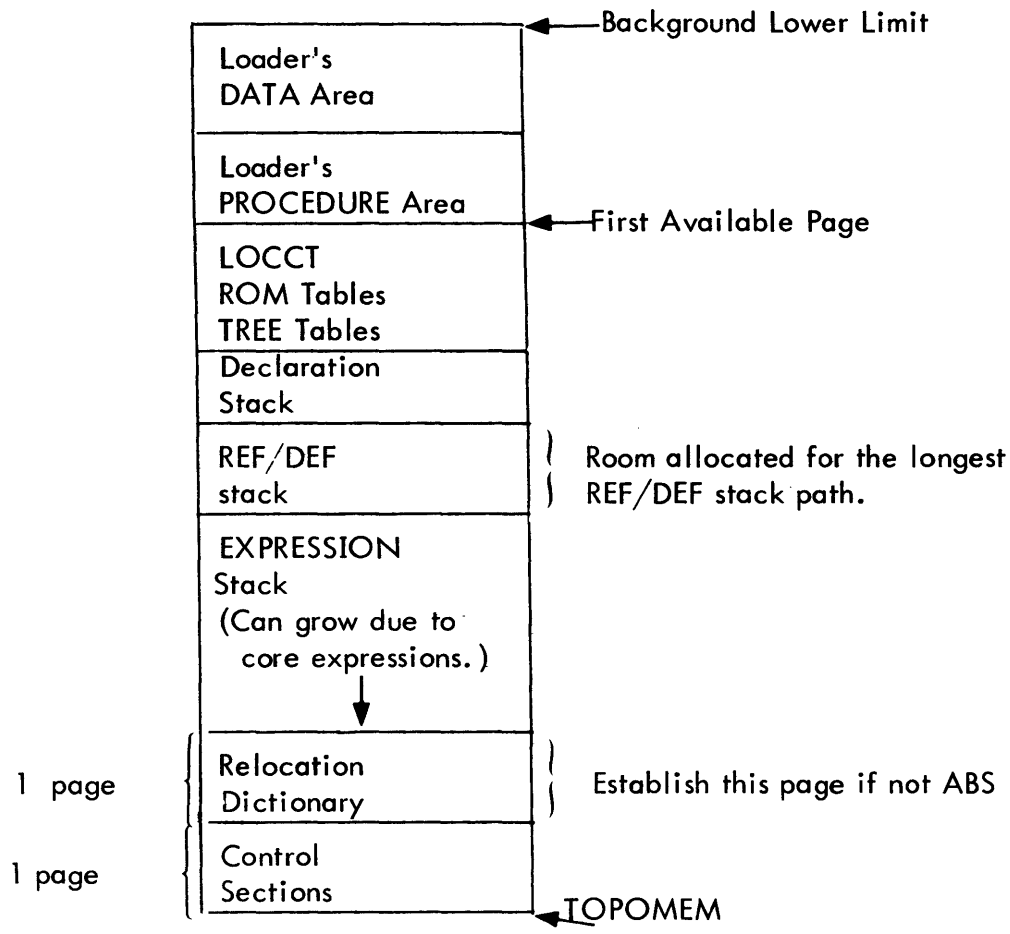


Figure 9a. How the Loader Uses Memory: Pass Two – Extended Memory Mode, Construction of Core Image Records

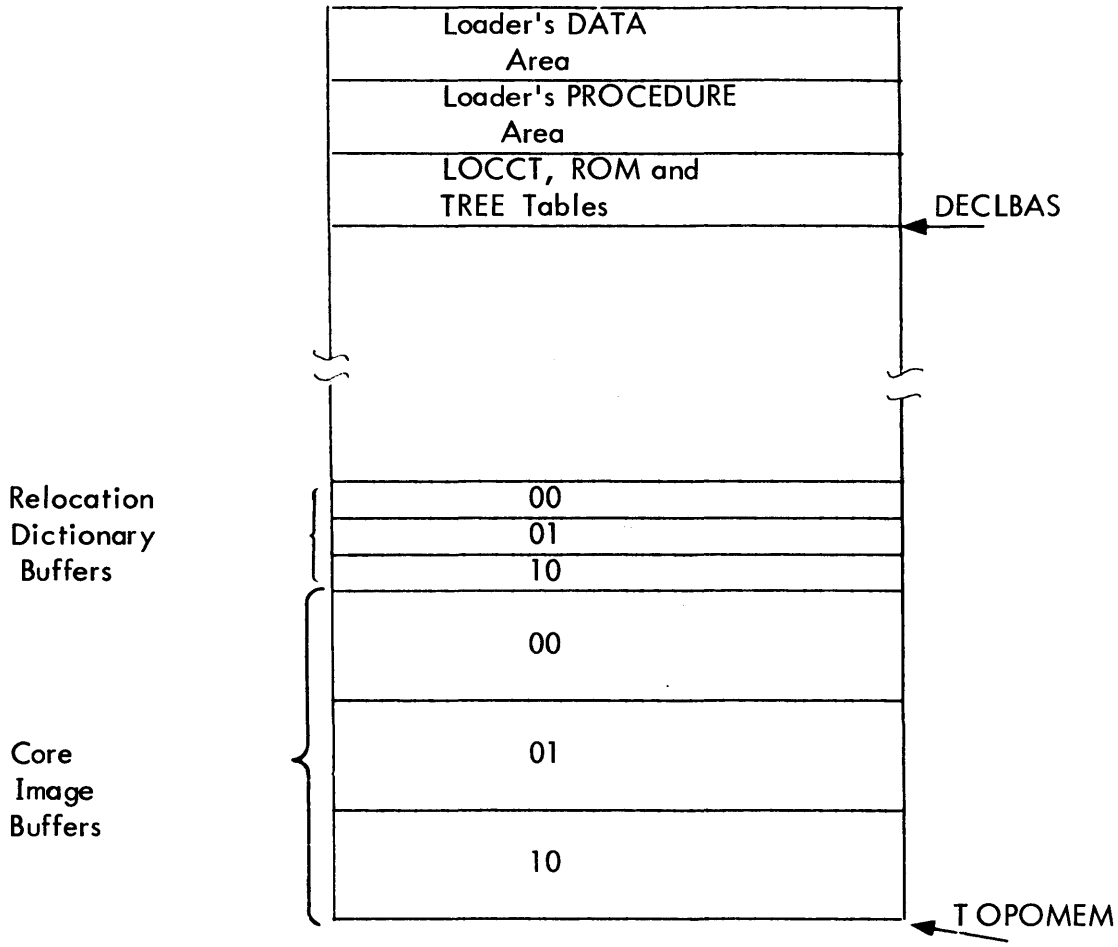


Figure 9b. How the Loader Uses Memory: Pass Two – Extended Memory Mode, Concatenation of Core Image Records

2.4 HOW THE LOADER OBTAINS MEMORY

2.4.1 Loader Running Under BPM

IN1 simply does a M:GP requesting the maximum (256) number of pages.

2.4.2 Loader Running Under UTS

Since memory must be obtained from both ends of the dynamic page area, and since UTS restricts the number of pages obtained, the above BPM technique does not suffice. IN1 initially gets four pages via M:GP. It then takes memory trap control (M:TRAP) such that "demand paging" is in effect. That is, whenever a memory violation occurs due to access of an unauthorized page, the Loader's TRAP routine (in the LDR module) is entered. TRAP computes the virtual page address requested and obtains the page via M:GVP.

2.5 MAINTAINING THE LOADER, DEBUG MODE

The Loader program, as a processor, cannot be executed as a user's program since it does not read its own control card. However, a special version of the Loader (called the "debug version") can run as a user's program, thus making Loader maintenance and modification an easier task. The debug version of the Loader is obtained by assembling LDR, IN1, and PS2 with the assembly parameter DEBUG EQU'd to 1. This causes code to be assembled which will read the LOCCT, ROM, and Tree Tables from a file (created by the LOCCT processor). It also causes the M:BI and M:DO DCBs to be built for reading the LOCCT and for handling !SNAPs, !MODIFYs, and !PMDs. A debug Loader can be assembled for either UTS or BPM, as determined by the parameter MODE.

Example:

```
1   !ASSIGN M:BI,(FILE,LOCCTTEST)
2   !RUN (LMN,DELOAD),(XSL,F)
3   !MODIFY ..................
:   !SNAP MESSAGE,MSG,(DECLSTK,DECLSTK+100),(+E200,+E600)
n   !PMD (00)
```

Card 1. The file must have been created by the LOCCT processor (see BPM Reference Manual, 90 09 54).

Card 2. The Loader (DELOAD in this example) must have been formed with LDR, IN1, and PS2 assembled in the DEBUG EQU 1 mode.

Card 3-n MODIFYs and debug commands.

In this mode the Loader may also be executed from the terminal under the RUN subsystem for BPM.

3.0 INPUT, OUTPUT, LOADER-GENERATED TABLES

3.1 INPUT

3.1.1 LOCCT, ROM, Tree Tables

Based on the !LOAD and !TREE cards, three related and contiguous tables are presented to the Loader upon entry: the Load Control Command Table (LOCCT), the Tree Table, and the ROM Table. If BPM is operating, the tables reside on sector 36 of the absolute area of the disk. Total size is contained in R6 upon entry to the Loader. If UTS is operating, the tables are left in core preceded by a word containing the size. A pointer to this area is in word JB:BCP of the JIT.

In either case, the Loader moves these tables into its first dynamic page (M:GP) during initialization.

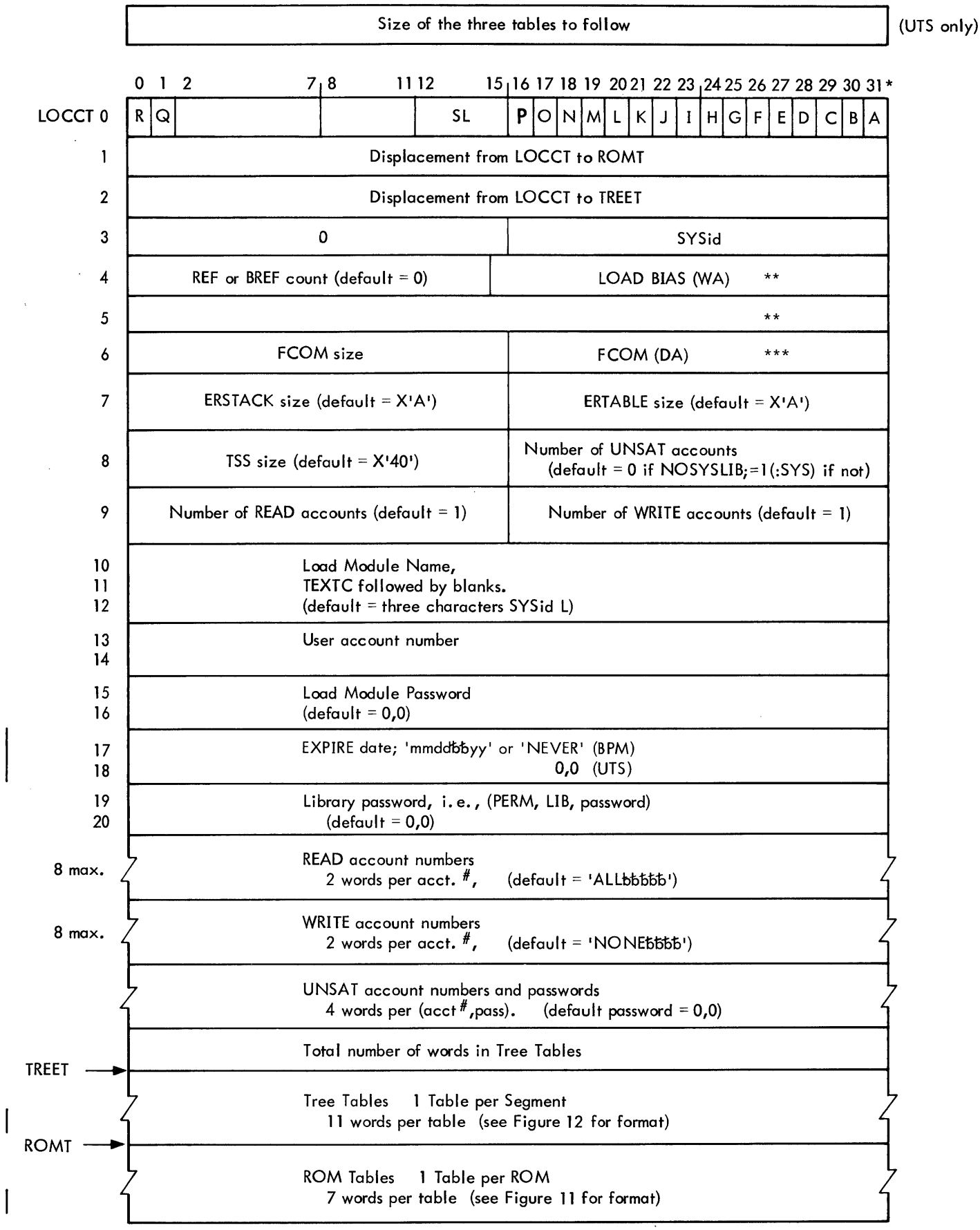
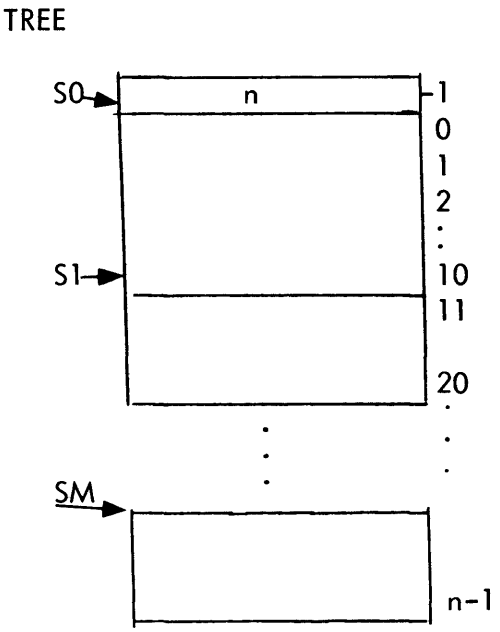


Figure 10. Loader Control Command Table (LOCCT)

Tree Tables

Overall picture for M segments (S0, ... SM)

n = total size of the tables



Tree Table Format (one 11-word Table per Segment)

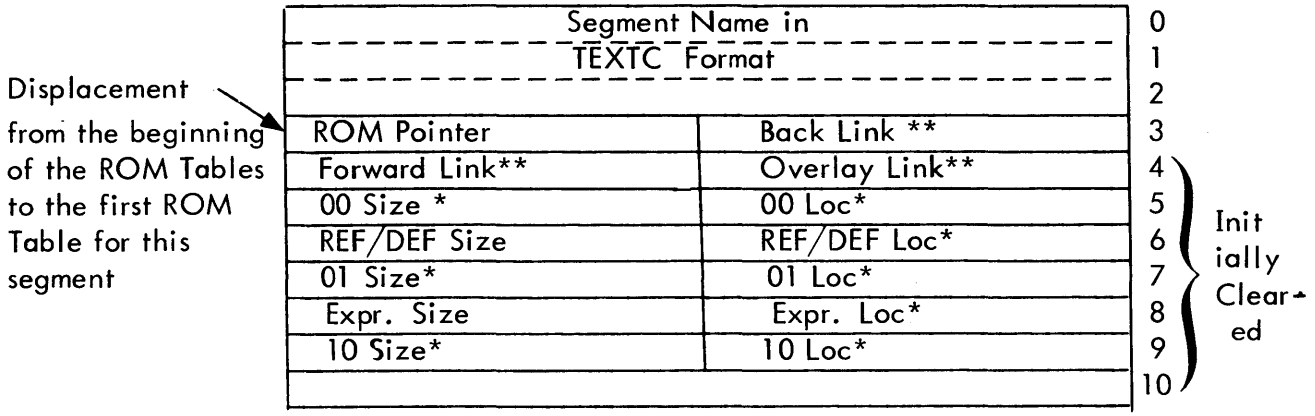


Figure 12. Tree Tables

Segment name is determined by the name of the first file in the segment. (If the load module has only one segment, i.e., the root, the keys begin with load module name. If no load module name was supplied, the name is idL.

Words 5-10 of each Tree Table are computed by the Loader.

Word 10 of the ROOT Tree Table is used to monitor the size of the REF/BREF Tables.

*Doubleword address or # of doublewords

** Displacements from TREE

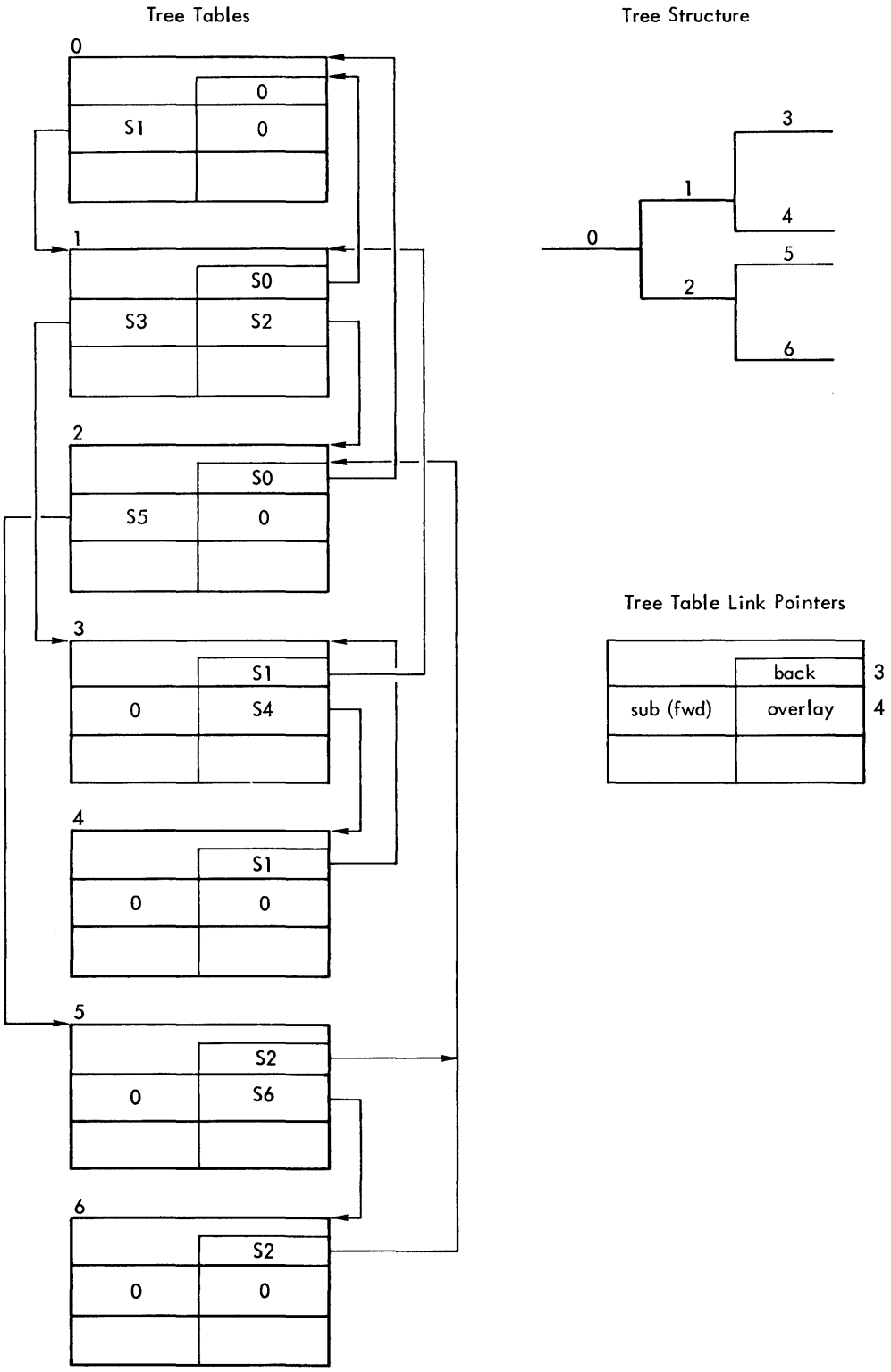


Figure 13. TREE Table Linking – in Relation to the Overlay Structure

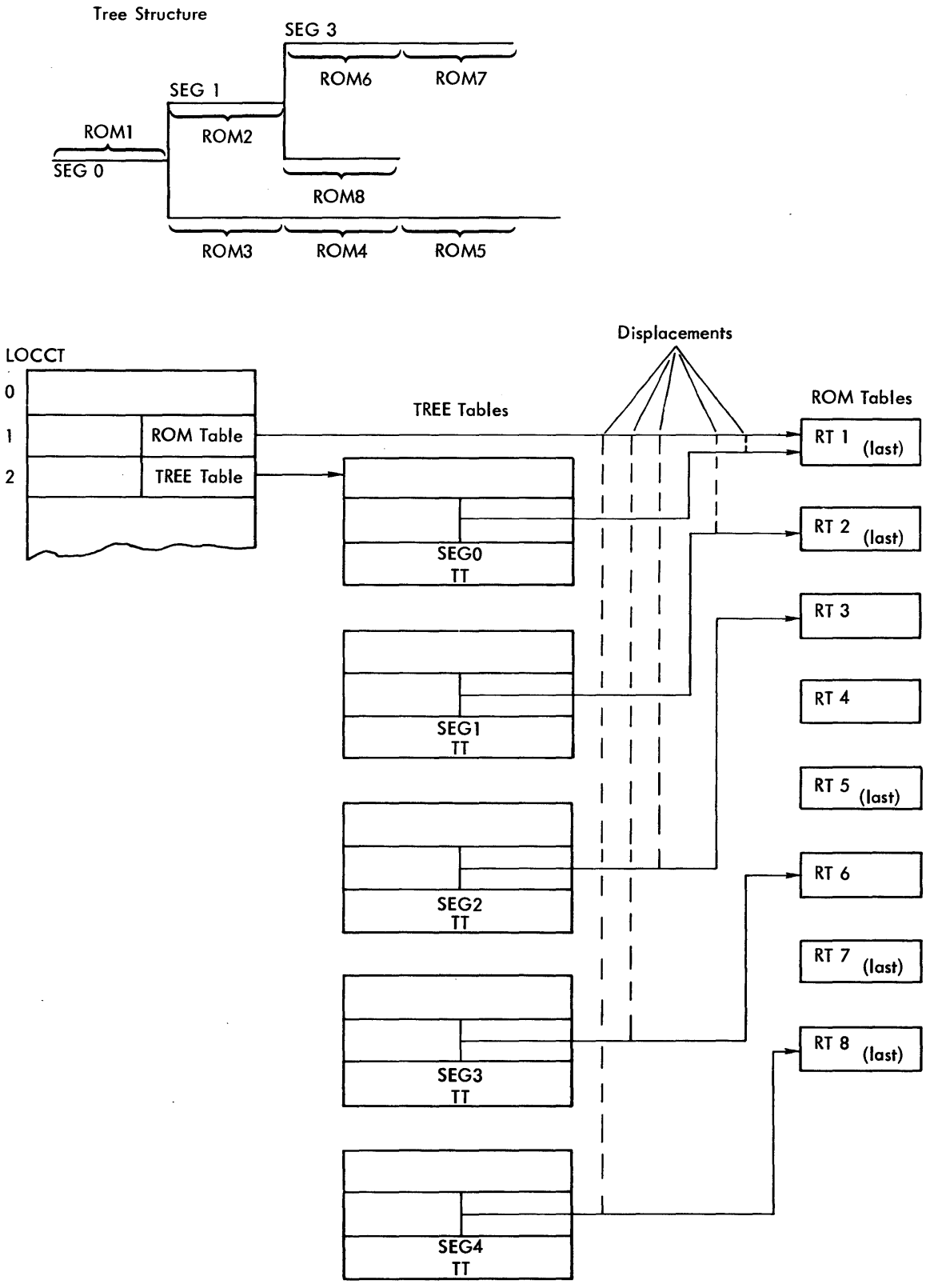


Figure 14. LOCCT, TREE, and ROM Table Relationships

3.1.2 Files (ROMs and Load Modules)

The Loader will access ROM files and load module files. All file names specified on the !LOAD card under the EF option appear in the ROM Table. Additionally, the files idB and idG may appear in the ROM Table if the user specified (BI) or (GO), respectively. During Pass One processing, the Loader augments the ROM Table by those library load modules which are to be included. Note that the Loader is entirely file-oriented. That is, ROMs coming from cards on BI will be read by CCI which creates a file by the name of idB. (Exception: M:EF may be assigned to labeled tape.)

a. ROM Files

A ROM file consists of one or more ROMs produced by an assembler or compiler (see the BPM Reference Manual (90 09 54) for a description of the ROM language). ROM files are accessed either from the accounts specified in the EF list (which the Loader sees in the ROM Table) or from the files idB and idG. (If a !TREE card has been included, idB and idG would appear in the ROM Tables for the root segment.)

b. Load Module Files

Load modules acceptable for combination with ROMs to form a new load module are built either by

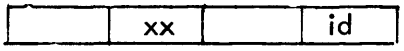
1) the Loader itself, in which case they are library load modules (see Output Section 3.2, for format);
2) PASS2 of SYSGEN; or
3) DEFCOM processor.

The HEAD of the input load module indicates one of the above sources in order that any attributes may be handled correctly. Any such load module must be of one protection type, relocatable, and not overlaid. Furthermore, if such a load module contains a DSECT, then the entire load module consists of that DSECT alone.

3.1.3 Registers and JIT Input

BPM

R6 = word size of LOCCT, ROM, and Tree Tables

R7 = 

SR1 = information needed by CCI or PASS3. This is simply stored and restored upon exit.

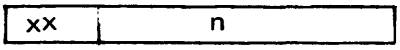
D4 = foreground COMMON bias

UTS

J:EUP = page number of the last user page

J:JIT, byte 3 = id

JB:BCP, byte 1 = page pointer to LOCCT Table -1. This word contains:



which is followed by the LOCCT, ROM, and Tree Tables.

SR1, D4 = same as BPM

xx = 0, if CCI called the Loader.
≠ 0, if PASS3 called the Loader.

n = size of LOCCT, ROM, and Tree Tables

id = system id (see Glossary)

3.1.4 ASSIGN Record

CCI builds a record of all ASSIGN information encountered during a job. The Loader examines this record to see if any F: number DCBs have been entered and, if so, will generate those DCBs with default entries (if they are PREFs within the user program). In UTS this record is read via an FPT code = X'2D'. In BPM the record is in section 35 of absolute area of the RAD and is read with an FPT code of X'16'. (See Section 16 of the F00 BPM Technical Manual for formats.)

3.1.5 Error Message File (ERRMSG)

This is a keyed file under the :SYS account. Its keys are of the form:

03	02	error number

The records are the text error messages. To alter the file, one uses the programs ERROM (Cat. No. CN706106) and ERRDATA (Cat. No. SI706107) for BPM, and the program ERRMWR for UTS.

When an error occurs, the Loader transfers control to MESSAGE (in LDR) with the error number in R3. MESSAGE builds the key, reads and prints the associated record.

Figure 15 is a listing of ERRMSG (to date).

KEY	MESSAGE	
020001	UNEXPECTED EOF	
020002	ILLEGAL RECORD I. D.	
020003	SEQUENCE ERROR	
020004	ILLEGAL RECORD SIZE	
020005	CHECKSUM ERROR	
020006	ABNORMAL I/O	
020007	CANNOT OPEN E. F.	
020008	STACK OVERFLOW	
020009	BIAS TOO LARGE	
02000A	ILL. ROM LANGUAGE	
02000B	BAD START ADDRESS	
02000C	UNEXPECTED ROM END	
02000D	REPEAT LOAD IS ZERO	
02000E	IMPROPER BOUND	
02000F	ILLEGAL ORG	
020010	BAD I/O RETURN FROM M:LM DCB	
020011	SEV. LEV. EXCEEDED	
020012	ILL. LIB. LOAD MOD.	
020013	NO ROOM TO ROUND DCBS TO PAGE BOUNDARIES. TRY FORCING XMEM	
020014	ILL. DSECT	
020015	ROOT SEGMENT TOO LARGE TO LOAD	IN2
020016	NEW UTS ERR FROM XMEM	IN2
020017	CANNOT ENTER XMEM. STACKS TOO LARGE.	IN2
020018	NOT ENOUGH ROOM TO CONCATENATE XMEM PAGES	IN2
020019	NO ROOM TO READ LIBRARY CORE IMAGE	EVL
02001A	NO ROOM TO READ LIBRARY RELOCATION DICTIONARY	WRT
02001B	NO ROOM FOR NEW EXPRESSION	WRT
02001C	NO ROOM TO BUILD DCB TABLE. TRY FORCING XMEM	WRT
02001D	NO ROOM TO BUILD DCB TABLE	WRT
02001E	LIBRARY LOAD MODULE REF/DEF STACK TOO LARGE TO UPDATE	WRT
02001F	INSUFFICIENT PHYSICAL MEMORY	
020020	BAD ASSIGN/MERGE RECORD	
020021	NO ROOM TO ADD LIBRARY LOAD MODULE TO ROM TABLE	
020022	NO ROOM TO READ LIBRARY REF/DEF STACK	
020023	NO ROOM TO UPDATE LIBRARY	
020024	INVALID KEY SUPPLIED FOR DELETE RECORD ON M:DIC	
020025	I/O ERROR ON M:DIC IN WRITESEG	
020026	ILLEGAL LIBRARY LOAD MODULE NAME	
020027	ABNORMAL I/O ON OPEN OR READ TO CORE LIBRARY	
020028	INVALID DECLARATION NUMBER REFERENCE (BAD ROM).	
020029	INVALID KEY SUPPLIED FOR WRITE RECORD ON M:DIC	
02002A	ILLEGAL LOADER TRAP	

Figure 15. ERRMSG File

KEY	MESSAGE
02002B	ABNORMAL I/O IN WRITELIB
02002C	CANNOT FIND REF/DEF NAME IN STACK
02002D	LIB LOAD MODULE TOO BIG - CANNOT USE EXTENDED MEMORY
02002E	LIB LMN IS NOT ALLOWED ON A PRIVATE VOLUME

Figure 15. ERRMSG File (cont.)

3.1.6 Modify File (idD)

This keyed file is built by CCI in the user's account on the basis of the !MODIFY cards.

Its keys are of the form:

TEXTC segment name concatenated with xx, where $0 \leq xx \leq n$ and
n = the hexadecimal number of !MODIFY cards.

See Section 16 of the F00 BPM Technical Manual for a more detailed format.

3.1.7 Core Libraries (UTS only)

Core libraries exist only under the :SYS account. An absolute copy of a core library's procedure area exists on swap storage associated with the name :Pnnn and is placed at run-time into a fixed area. The DEFs for :Pnn which relate the core library's context area (preceding the user's blank COMMON) with the user and the library procedure are contained in a load module (formed by DEFCOM) named :Pn. The Loader's job is to read :Pn, merge the DEFs into the REF/DEF stack of the target load module, and signal the !RUN processor that it is to associate :Pnn with this program. The signal consists of placing the text :Pnnn in the HEAD record of the load module.

:P0 is the name of the FORTRAN core library with debug.
:P1 is the name of the FORTRAN core library without debug.

See Chapter 6 of the UTS System Management Guide for details on core libraries.

3.2 OUTPUT

3.2.1 Load Modules, Overall Format

A load module is a keyed file whose name was supplied on the !LOAD card (default = idL). The keys and records are as follows:

a. Key = HEAD

Record

BPM

	0	8	16	24	31
0	8X	00	FF	n	
	A B	SL		START address	
2	TCB*		Module Bias*		
3	DATA (00) Base*		PROCEDURE (01) Base*		
4	STATIC DATA (10) Base*		Next Available Page*		
5	MAX RF/DF SIZE		TREE Size		

UTS

0	8X	00	FF	n	
1	A B	SL	START address		
2	TCB*		Module Bias*		
3	DATA Size*		DATA (00) Base*		
4	PROCEDURE SIZE *		PROCEDURE (01) Base*		
5	MAX RF/DF Size		TREE Size		
6	DCB Size*		DCB Base (10)*		
7	0		0		**
8	0		0		
9	0		0		***
A	0		0		
B	0		0		

(Footnotes are on next page.)

Footnotes to keys and records shown on previous page:

*Doubleword address

In byte 0, word 0

X = 0, load module produced by Loader.
= 1, load module produced by SYSGEN.
= 2, library load module produced by Loader.
= 3, load module produced by DEFCOM (consists of HEAD, TREE, and REF/DEF (Stack).
= 5, pages load module produced by Loader.

n = number of bytes in the HEAD record. For UTS, n = X'30'; for BPM, n = X'18'.
A = 1, abs module
B = 1, NOTCB
SL = Final Severity Level

** Word 7 — If DEFCOM output, this word = byte size of DATA area.
*** Words 9, A, B — If the LMN is associated with a core library, these words are :Pnnn in TEXTC format.

b. Key = TREE — Record is the Tree Tables (see Figure 12).

c. Segment Components - Standard Load Module

For each segment, the following records are built:

	Key	Record
Segment Name Concatenated with:	00	REF/DEF stack
	01	EXPR stack
	02	00 REL DICT
	03	00 Control Sections
	04	01 REL DICT
	05	01 Control Sections
	06	10 REL DICT
	07	10 Control Sections

d. Segment Components - Paged Load Module

For each segment, the expression stack and REF/DEF stack records have the same format as those for the standard load module. Relocation dictionary records are not constructed.

The core images are partitioned into records of at most 512 words in length with 3-byte keys of the following format:

SEG	00	PAGE

where SEG = the TREE segment number of the segment containing the core image.

PAGE = the page number of the virtual page that will contain this record at execution time.

All core image records are one page in length except for the first record of an overlay segment's 00, 01, and 10 areas. The length of this record satisfies the following: at execution time, the record begins at the execution bias for this protection type and ends at the next page boundary.

3.2.2 Library Load Modules

A library constructed by the Overlay Loader consists of two keyed files, :LIB and :DIC. The library load modules actually reside in one file (:LIB). :DIC is a dictionary whose keys are the text names of DEFs. The record associated with a dictionary key is the text name of the load module (within :LIB) in which that DEF is defined. Thus, in order to locate the unique group of records within :LIB which pertain to a given PREF, the Loader does a keyed READ to :DIC, the key being the PREF which is being satisfied. This keyed READ returns the library load module name within :LIB. With this information the Loader can then read the library load module records into core and merge them with the target load module.

The keys and records in :LIB are identical to those of non-library load modules (see above) except that the keys "HEAD" and "TREE" are concatenated with the TEXT load module name (to keep them unique). Each individual library load module name is "synonymous" (in a file sense) with the name :LIB.

A slight difference also exists in the REF/DEF and expression stack formats. The VALUE word of an entry in the REF/DEF stack is actually the head of a chain through the expression stack of all those entries which involve that REF/DEF. (This expedites subsequent merging of the stacks when the library is included in a user program.)

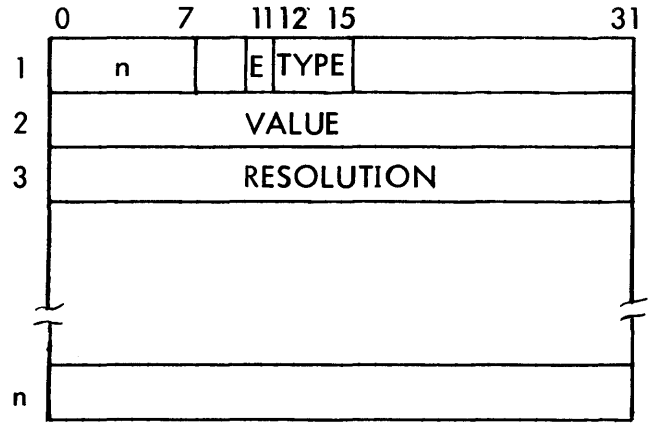
3.2.3 REF/DEF Stack

There is one REF/DEF stack for each segment. A REF/DEF stack is composed of entries for every control section and forward reference in the segment. It also contains an entry for every name (DEF, REF, SREF) in the segment which does not occur in this segment's backward path.

Before a name is added to a segment's REF/DEF stack, the segment's stack and the REF/DEF stacks for this segment's backward path are searched. If the name is not in these stacks, a new entry is added to the segment's stack. If the name already exists, the entry in which the name appears is treated as follows:

New Name	Type of Existing REF/DEF Entry	Modification of Existing REF/DEF Entry
DEF	DEF	Double DEF
DEF	REF	DEF
DEF	SREF	DEF
REF	DEF	Used DEF
REF	REF	No change
REF	SREF	REF
SREF	DEF	Used DEF
SREF	REF	No change
SREF	SREF	No change

GENERAL REF/DEF STACK FORMAT



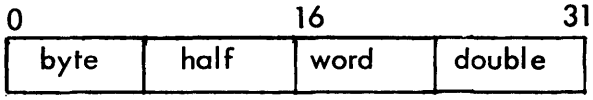
where:

n = number of words in this entry.
E = 1, if the entry has a VALUE

TYPE =
- 0 or 8 DEF
- 1 SREF
- 2 PREF
- 3 or B Dummy Section
- 4 or 6 Control Section
- 5 or 7 Forward Reference

VALUE = constant or address if the load module is not a library
or
head of a chain in the expression stack if the load module is a library (see SQZ, Section 7.0).

RESOLUTION = the resolution in which the VALUE is expressed. Resolution is of the form:

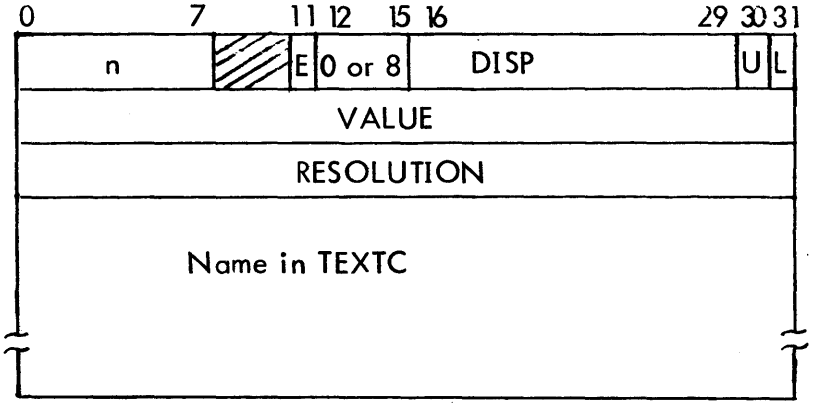


If the VALUE is a constant, the RESOLUTION word is 0.

If the VALUE is an address, one and only one byte of the RESOLUTION word is nonzero (viz., the appropriate byte = X'01').

If the RESOLUTION assumes a form different from either of the above, the VALUE is of mixed resolution. (In this case the load module cannot be relocated and is forced ABS.)

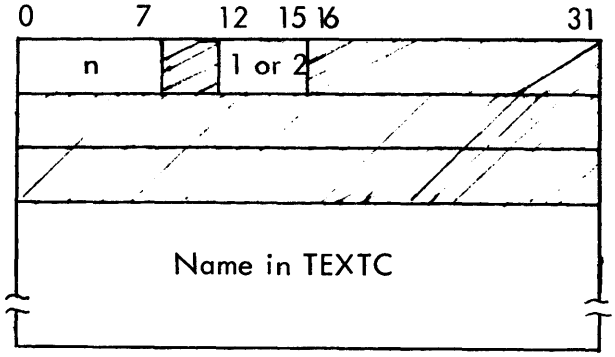
TYPE = 0 or 8 (DEF)



where:

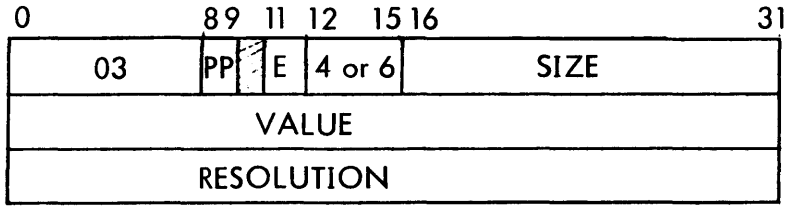
TYPE = 0, this entry is a DEF.
= 8, this entry is a double DEF.
E = 1, the DEF has a value.
DISP = Displacement to the segment in the Tree Table where the DEF is located.
U = 1, used DEF (the DEF has been referenced).
L = 1, the DEF was defined in a library.

TYPE = 1, 2 (SREF or REF)



TYPE = 1, SREF
= 2, PREF

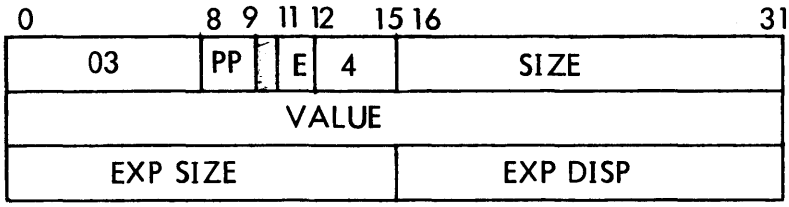
TYPE = 4 or 6 (Control Section)



where:

TYPE = 4, when first declared in PASS1 (LP1).
= 6, after rereading the declaration in PASS2 (LP1 of EVL).
E = protection type
SIZE = size of the control section in doublewords.

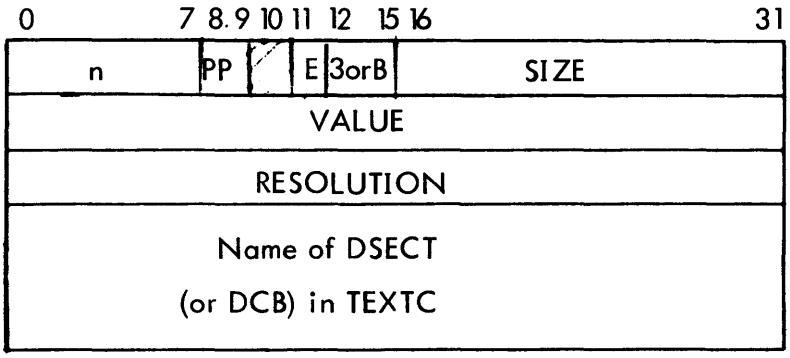
NOTE: A special entry is created by the Loader and inserted in front of a library load module's REF/DEF stack. It has a TYPE = 4, but can be detected (in PASS2) because all previous control sections would have been changed to 6 by this time.



where:

SIZE = Size of the load module's core image in doublewords.
VALUE = Location of this load module's core image (within the target load module).
EXP SIZE= Word size of the load module's expression stack.
EXP DISP= Displacement of load module's expression stack within this segment's expression stack.

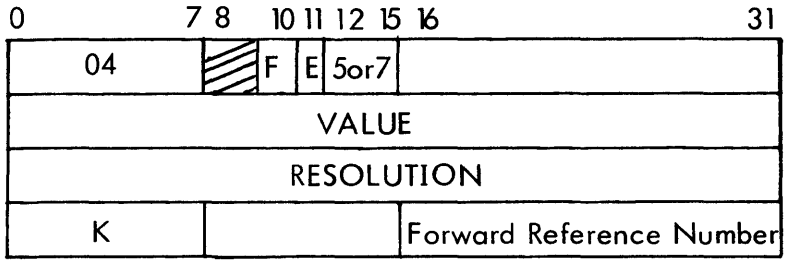
TYPE = 3 or B (Dummy Control Section)



where:

TYPE = 3, Dummy Control Section
= B, At the end of PASS1, all PREFs (TYPE2) with names beginning with M: or F: are changed to TYPE B, indicating that the Loader is to build them at the end of the second pass.

TYPE = 5 or 7 (Forward Reference)



where:

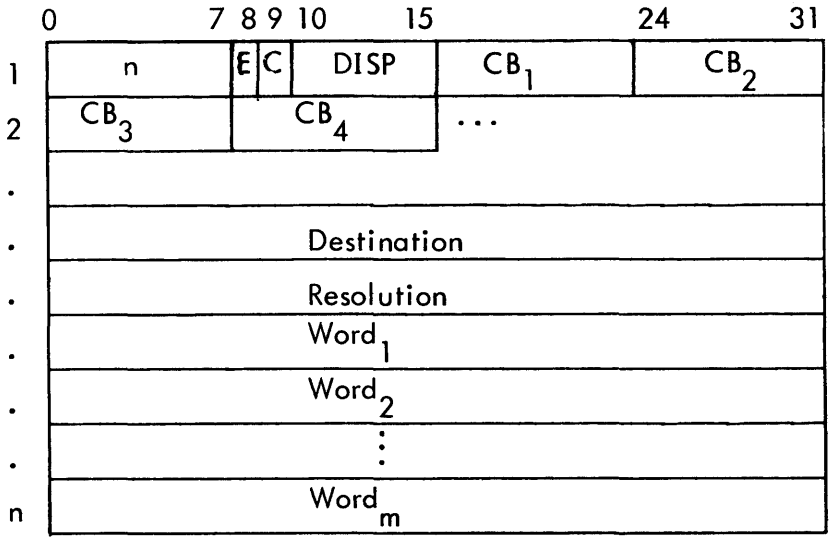
TYPE = 5, Forward Reference.
= 7, Forward Reference is defined from a library.
K = 0, Until the forward reference is defined.
=FF, Define forward REF and "release" the reference number.
=F0, Define forward REF and hold the reference number until module end.
F = 1, the forward reference is used in a "Define forward reference and hold" expression.

3.2.4 Expression Stack

The Loader builds an entry in the expression stack by re-formatting a ROM expression. This re-formatting process consists of grouping all of the control bytes together in one part of the entry, and all of the operands in another. If the ROM operand is a constant, it is transferred verbatim from the ROM to the operand portion of the entry. If the ROM operand is a declaration number, the REF/DEF stack pointer is accessed from the declaration stack and placed in the operand portion of the expression entry. If the ROM operand is a forward reference number, the corresponding REF/DEF stack pointer is transferred to the operand portion of the entry. Some control bytes have no operands (viz., expression end or change resolution) and therefore, have no corresponding item in the operand portion. Thus, the control byte portion of the entry is related sequentially to the operand portion, except in the case where no operand exists.

The value of an expression is deposited either in a REF/DEF stack entry or in a field in the core image of the target load module. (See Section 2.1.1f). In the first case, the destination of the expression's value is described by a pointer to the entry in the REF/DEF stack. In the second case, the destination is described by a core expression. A core expression contains the field size, in bits (which can cross up to eight words of the core image); the address of the last word in the core image to be changed; and the terminal bit position of the field.

GENERAL EXPRESSION STACK ENTRY



where:

n = number of words in entry

E = 1, this entry has been evaluated.
= 0, this entry has not been evaluated.

C = 0, this entry's Destination is a pointer to the REF/DEF stack.
= 1, this entry's Destination is a core expression.

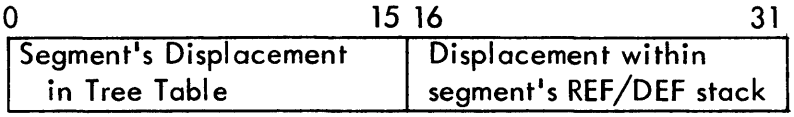
DISP = number of words to Word 1.

Destination: (where the value of the entry is to be deposited) =

one of the following forms, depending upon the value of C.

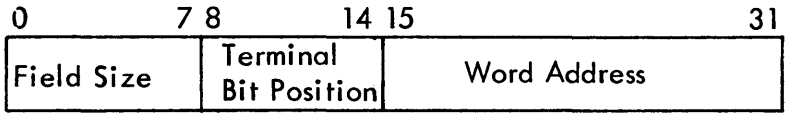
REF/DEF Pointer

If C = 0



Core Expression

If C = 1



Resolution: Same as REF/DEF stack.

CB_i = a control byte of the expression.

$Word_i$ = is referenced by a control byte and is a constant or pointer to the segment's REF/DEF stack (same form as Destination where C=0).

3.2.5 Relocation Dictionary

If ABS is not specified on the !LOAD card, each segment will have records of relocation dictionaries (one per protection type). One relocation digit is developed for each word in the protection area.

Relocation Dictionary Digits

Digit	Type of Relocation
0	relocate the word at byte resolution.
1	relocate the word at halfword resolution.
2	relocate the word at word resolution.
3	relocate the word at doubleword resolution.
8	relocate the left half of the word at doubleword resolution.
9	relocate the right half of the word at doubleword resolution.
A	relocate both halves of the word at doubleword resolution.
E	absolute.

Notice that relocation digits exist only for items that terminate on halfword boundaries.

A load module which has an item not amenable to one of these digits is set to ABS.
Example:

```
        BOUND  4
ZAP     EQU    DA($)
        GEN, 8, 16, 8   0, ZAP, 0
```

or

```
        BOUND  4
ZAP     EQU    $
        GEN, 3, 17, 12  0, ZAP, 0
```

Either of these would cause the module to be set ABS since ZAP does not terminate on a halfword boundary.

3.2.6 Miscellaneous (Map, Diagnostics, Severity Level)

The map, diagnostics, and the severity level of the load module are output via the M:LL DCB (normally the printer):

a. Load Map

The load map is generated at the end of the load process. For each segment, the map includes:

i) A header consisting of the segment name and size. For the root segment, the load module name, account number, start address, and bias are also listed.

ii) A summary of the segment's protection type boundaries and sizes of the format:

```
****PROTECTION TYPES: 00  DATA        01 PROCEDURE   10 STATIC
                      SEGHI-0 valhi   SEGHI-0 valhi   SEGHI-2 valhi
                      SEGLO-0 vallo   SEGLO-1 vallo   SEGLO-2 vallo
                      00SIZE=size     01 SIZE=size    10 SIZE=size
```

where valhi = the high word address for this protection type.

vallo = the start address (word resolution) for this protection type.

size = the size, in words, of the protection type area.

iii) A list of any unsatisfied primary references (PREFs).

iv) A list of any unsatisfied secondary references (SREFs).

v) A list of any multiply-defined definitions (DDEFs).

vi) A list of definitions with absolute values (ADEFs).

vii) A list of relocatable definitions and control sections for this segment, sorted either by value or by name. A value sort produces a list of the DEFs and control sections in increasing value, with a new line started for a CSECT or DSECT. The control section's address and protection type is noted in the left-hand margin of this line and its size is noted in the right-hand margin.

A name sort really produces two lists. The first list, entitled 'SECT-PROGRAM SECTIONS MAP' contains the control sections (in increasing value) and the first DEF in each section. One (lowest in value) control section, its first DEF, and the control section size is printed on a single line. The second list, entitled 'RELOCATABLE DEFINITIONS SORTED BY NAME', lists the DEFs, sorted alphanumerically by name over the entire segment.

In both the value and name type of DEF lists, the control sections are printed in the format:

$$\text{value}\begin{Bmatrix}\text{CSECT}\\ \text{DSECT}\end{Bmatrix}\text{p}$$

where p = the protection type of the control section.

value = the word address at which this section begins.

The relocatable DEFs have the format:

value r symbol

where value = the value of the definition, expressed as a word address.

r = the byte displacement (i.e., the two high order bits of the value if it were expressed as a byte address).

symbol = the symbolic name of the item.

The following flags can precede the symbolic name of a DEF (or ADEF).

* = unused definition.

+ = multiply defined definition.

- = definition satisfied from a library.

The map for each segment starts on a new page. For the lists (iii)-(vii), four symbols are listed on a line unless there is a large symbol which cannot fit in one column. In this case the symbol is printed on a single line. Lists (iii)-(vi) are always sorted by name.

b. Diagnostics

The diagnostic consists of the pertinent record obtained from the ERRMSG file and the following information: the name of the element file currently being processed, the sequence number of the record most recently read, and a third field of data pertinent to the particular error that occurred. (See Figure 15b for a list of the error message keys and the corresponding data printed in this field.)

c. Severity Level

A nonzero severity level is printed at the end of the load process immediately before the map is printed. The final severity level is actually the maximum of any severity levels inherited from the ROMs and those generated internally by the loader.

Internal Loader-generated Severity Levels:

Type of Error	Severity
PREF	7
DDEF	4
REF load table exceeded	F
BREF load table exceeded	6

(After printing the final severity level, it is compared with the maximum specified by the user (for CCI). If it is greater loading is aborted).

d. Register Output for PASS3

D4 = 0 if normal return.

= -1 if abnormal return.

SR1 = original contents upon entry to the Loader.

ERROR KEY	Diagnostic Information Output
020001	SR3
020002	Record I.D.
020003	(none)
020004	Record Size
020005	(none)
020006	SR3
020007	SR3
020008	SR3
020009	Bias
02000A	Object Module Control Byte
02000B	Start Address
02000C	(none)
02000D	(none)
02000E	Byte addr of load relocatable destination
02000F	SR4 (for debugging purposes)
020010	SR3
020011	Computed Severity Level
020012	(none)
020013	No. of words to be added to 10 area
020014	1st 4 characters of DSECT name
020015	No. of words exceeding available background
020016	(none)
020017	No. of words that stacks exceed available background
020018	No. of words exceeding available background
020019	No. of words in library's core image and rel. dict.

Figure 15b. Variable Diagnostic Information

ERROR KEY	Diagnostic Information Output
02001A	Size of relocation dictionary
02001B	(none)
02001C	(none)
02001D	No. of words in DCB Name Table and its rel. dict.
02001E	(none)
02001F	Register 0
020020	SR3
020021	High addr. of REF/DEF stack (which would overwrite exprstk)
020022	Size of library load module's REF/DEF stack
020023	Size of REF/DEF stack corresponding to old version of library lmn
020024	Key Size
020025	SR3
020026	No. of characters in load module name
020027	SR3
020028	Invalid Declaration Number
020029	Key Size
02002A	Register 0
02002B	SR3
02002C	1st 3 characters and byte count of name
02002D	(none)
02002E	(none)

Figure 15b. Variable Diagnostic Information (cont.)

DESCRIPTION OF COMMON LOADER ERROR MESSAGES

UNEXPECTED EOF	An end-of-file was encountered before the end of an object module was reached (incomplete object module).
ILLEGAL RECORD I.D.	The type of record read was neither X'3C' nor X'1C' (object module), nor X'81', X'82', or X'83' (Load Module).
SEQUENCE ERROR	The cards of an object module were out of sequence.
ILLEGAL RECORD SIZE	The number of bytes in an object module card was less than four or greater than X'6C'.
CHECKSUM ERROR	A bit (or bits) was dropped in punching or reading the object module.
ABNORMAL I/O	An abnormal return was encountered while reading a library load module.
CANNOT OPEN E.F.	An element file could not be opened. (It does not exist, it has a password, etc.)

STACK OVERFLOW (BPM)	Insufficient memory in which to load. If no map has been partially printed, the module is too large. If a map has been partially printed, some unsatisfied primary references have caused the stacks to grow to excessive size.
BIAS TOO LARGE	At the given bias, the load module will exceed 131K of memory.
ILL. ROM LANGUAGE	The object language in a relocatable object module was not translatable (assembler or compiler error).
BAD START ADDRESS	A start address was given which is either not on a word boundary or is not within the load module.
UNEXPECTED ROM END	Module end was given on some card of the object module other than the last card (assembler or compiler error).
REPEAT LOAD IS ZERO	An assembler or compiler generated a repeat load item with a 0 count (assembler or compiler error).
IMPROPER BOUND	A short- or long-form relocatable item was not on a word boundary.
ILLEGAL ORG	An origin was generated having no resolution or was not within load module (assembler or compiler error).
BAD I/O RETURN FROM M:LM DCB	The load module file could not be opened.
SEV. LEV. EXCEEDED	The severity level specified in the LOAD card was less than that encountered in some object module or that generated by the Loader (a DDEF yields a severity level of 4, a PREF yields 7).
ILL. LIB. LOAD MOD.	(PERM, LIB) was specified and the load module had one of the following: 1. More than one protection type. 2. No relocation dictionary (ABS was specified or forced by the Loader due to nonstandard relocatable fields). 3. More than one segment.
ILL. DSECT	Two dummy sections having the same name but different protection types were encountered.
INSUFFICIENT PHYSICAL MEMORY (UTS ONLY)	This message can only occur running under UTS and has the same meaning as STACK OVERFLOW for BPM.

3.3 LOADER-GENERATED TABLES

All Loader-generated tables reside in the root segment of the load module in the order indicated by Figure 1. Loader-generated tables are the TCB, Tree Tables, DCB Name Table, REF/BREF Tables, and DCBs.

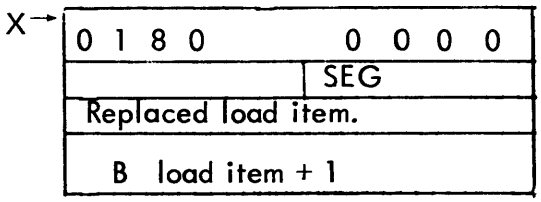
3.3.1 Formats for the TCB and DCB Name Table are in the BPM Reference Manual. The TCB resides in 00. The DCB Name Table resides in 01 for BPM and 10 for UTS.

3.3.2 TREE. A copy of the Tree Tables (see Figure 12) is placed at the beginning of the 01 area (as well as being separately recorded in the TREE record).

3.3.3 REF/BREF Tables

REF mode

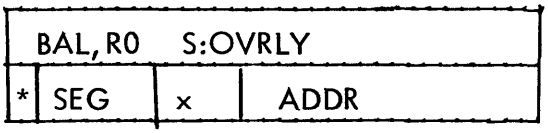
An entry is created for every load item involving a REF defined in a higher segment. The load item is replaced by a CAL1,8 X where X is the REF Table entry address (a PLIST for the CAL).



SEG = 17 bit address of higher segment name in Tree Table.

BREF

An entry is created for every branch type instruction involving a REF to a higher segment. The branch type instruction is replaced by a branch (of the same type) to the BREF entry.



where: S:OVRLY is a system library routine

SEG = segment number (Tree Table displacement/11)

ADDR = address field of replaced instruction

*, x = indirect and index fields from replaced instruction

EXAMPLE:

Assume that a segment S references ZAP (defined in a higher segment):

```
Segment S      .
               .
               .
               REF      ZAP
               .
               .
               .
           α   BAL, 7   *ZAP
               .
               .
               .
```

If REF loading mode:

```
           α   CAL1, 8   β
               .
               .
               .
```

β

0 1 80	0 0 0 0
	SEG
BAL,7	*ZAP
B	α + 1

SEG is as defined for REF above.

If BREF loading mode:

α BAL,7 β

β

BAL,R0			S:OVRLY
1	SEG	0000	ZAP

SEG is as defined for BREF above.

3.3.4 DCBs

The Loader will build a DCB if, at the end of PASS1, there exist any PREFs which begin with M: or F:. This can occur if: 1) CCI's ASSIGN record contained F: number entries; 2) the user had a REF DCB name and had no ROMs or libraries which satisfied this REF; 3) the NOTCB option is absent, whereupon an M:DO is generated; 4) a !TREE card is present, whereupon an M:SEGLD is generated.

All Loader-generated DCBs are DSECTs whose allocation is forced to the root. The standard 22 words are allocated for the fixed portion of the DCB. In the variable length parameter portion of the DCB, three words are allocated for file name, two words for account, two words for password, three words for INSN numbers, and three words for OUTSN. Two additional words are allocated for an EXPIRE date for UTS DCBs. The

total DCB size is 48 words for BPM, 51 words for UTS. Default information is placed into recognized DCB names. The recognized DCB names and their defaults are shown in Figure 16.

DCB NAME	FUNCTION	RECORD BYTE SIZE	OPERATIONAL LEVEL
M:C	Input	120	C
M:OC	Input/Output	85	OC
M:LO	Output	132	LO
M:LL	Output	132	LL
M:DO	Output	132	DO
M:PO	Output	80	PO
M:BO	Output	120	BO
M:LI	Input	120	LI
M:SI	Input	80	SI
M:BI	Input	120	BI
M:SL	Output	132	SL
M:SO	Output	80	SO
M:CI	Input	120	CI
M:CO	Output	120	CO
M:AL	Output	80	AL
M:EI	Input	120	EI
M:EO	Output	120	EO
M:GO	Output	120	NO
F:101	Input	0	OC
F:102	Output	0	OC
F:103	Input	0	PR
F:104	Output	0	PP
F:105	Input	80	SI
F:106	Output	120	BO
F:108	Output	132	LO

Figure 16. Recognized DCBs and their Defaults

For UTS, nonstandard DCBs (i.e., those not listed in Figure 16) are assigned to 'ME', which goes to the terminal for an on-line user or to device 'NO' for batch.

3.4 EXAMPLES

The following example is designed to illustrate: 1) a load module's expression stack in relation to its REF/DEF stack, and 2) the correspondence of these two stacks to the ROM from which they were derived. This example should also clarify many of the files and tables discussed in this chapter.

3.4.1 A Sample Program

The following program was assembled under the METASYM processor.

```
1                                                   SYSTEM          SIG7FDP
2                                                   DEF             AB1
3                                                   REF             AB2
4       01  00000    6A900000  X   START            BAL,9           AB2
5       01  00001     00000008  02                  DATA            ZAP+2
6       02  00000                                   CSECT           0
7       02  00000                                   RES             5
8       02  00005     000000FF  A   AB1              DATA            X'FF'
9               02  00006                 ZAP       EQU             $
10              01  00000                           END             START
CONTROL SECTION SUMMARY:  01  00002  PT 0   02  00006  PT  0
```

3.4.2 The ROM

Following is a load-item-by-load-item interpretation (known as a ROMBUST) of the ROM for this program. The load items are interpreted in the order that they were output by the METASYM processor. Note that each load item is listed, in hexadecimal, on the line immediately above its verbal description.

ROMBUST OF SAMPLE PROGRAM

```
RECORD NUMBER: 0
RECORD TYPE: LAST, MODE: BINARY, FORMAT: OBJECT LANGUAGE.
SEQUENCE NUMBER 0
CHECKSUM: 200
RECORD SIZE: 66

0303C1C2F1
DECLARE EXTERNAL DEFINITION NAME (3 BYTES) NAME: AB1 DECLARATION
    NUMBER : 1


0503C1C2F2
DECLARE PRIMARY REFERENCE NAME (3 BYTES) NAME: AB2 DECLARATION NUMBER
   2

0C000008
DECLARE NONSTANDARD CONTROL SECTION DECLARATION NUMBER: 3
ACCESS CODE: FULL ACCESS. SIZE 8 X'8'

0C000018
DECLARE NONSTANDARD CONTROL SECTION DECLARATION NUMBER: 4
ACCESS CODE: FULL ACCESS. SIZE 24 X'18'


0A0101000000142004 02
DEFINE EXTERNAL DEFINITION
NUMBER 1
ADD CONSTANT: 20 X'14'
ADD VALUE OF DECLARATION (BYTE RESOLUTION)
NUMBER 4
EXPRESSION END


04200302
ORIGIN
ADD VALUE OF DECLARATION (BYTE RESOLUTION)
NUMBER 3
EXPRESSION END


826A900000
LOAD RELOCATABLE (SHORT FORM). RELOCATE ADDRESS FIELD (WORD RESOLUTION)
RELATIVE TO DECLARATION NUMBER 2
THE FOLLOWING 4 BYTES: X'6A900000'
```

```
8400000008
LOAD RELOCATABLE (SHORT FORM), RELOCATE ADDRESS FIELD (WORD RESOLUTION)
RELATIVE TO DECLARATION NUMBER  4
THE FOLLOWING 4  BYTES:  X'8'

040100000014200402
ORIGIN
ADD CONSTANT:  20 X '14'
ADD VALUE OF DECLARATION (BYTE RESOLUTION)
NUMBER 4

EXPRESSION END


44000000FF
LOAD ABSOLUTE THE FOLLOWING 4 BYTES:  X'000000FF'

0D220302
DEFINE START
ADD VALUE OF DECLARATION  (WORD RESOLUTION)
NUMBER  3
EXPRESSION  END


0E00
MODULE END
SEVERITY LEVEL:  X'0'
```

3.4.3 The Load Module

The following load card was used to form a load module for this program:

!LOAD (EF, (SAMPLE)), (NOTCB), (SL,A), (LMN,TARGET)

(Where the ROM was located in the file with name SAMPLE).

The resultant load module is listed below.

TARGET LOAD MODULE

```
HEAD

00    8000FF18 47006F00 00003700 37003800 39003900   0011000C

07E3C1D9C7C5E300
00    03160000 00006E00 00000100 04100000 0001B81C   01000000 03C1C2F1 04020000
08    00000000 00000000 03C1C2F2 03160001 00006E00   00000100 03160003 00006F02
10    00000100

07E3C1D9C7C5E301
00    06840120 02000003 00000003 01000000 00000014   0000000E 04432202 113E6F00
08    00000000 00000007

07E3C1D9C75E302
00000 E2EEEEEE

07E3C1D9C7C5E303
00    6A900000 00006E0A 00000000 00000000   00000000 00000000 00000000 000000FF

07E3C1D9C5E304
00    2EEEEEEE 9E9E99EE EEEEEE

07E3C1D9C7C5E305
00    00000000 00000000 0000000C 06E3C1D9 C7C5E301 40404040 00000000 00000000
08    00043700 00113E38 000B3800 000A3E53 00003900 00000000 00000000 00000000
10    00000000 06040120 02000003 00000003 00000000 00000014

TREE
00    0000000C 06E3C1D9 C7C5E305 40404040 00000000   00000000 00043700 00113E38
08    000B3800 000A3E53 00003900 00000000
```

3.4.4 The Relationship Between the Expression Stack and the REF/DEF Stack

The REF/DEF stack of the preceding load module (the second record listed) has entries as follows:

TYPE	DISPLACEMENT FROM STACK BASE
Control Section	Word 0
DEF (of AB1)	Word 3
PREF (of AB2)	Word 7
Control Section	Word B
Control Section	Word E

The first REF/DEF entry is a special control section and corresponds to Declaration Number 0 (for one-pass assemblers and compilers). The subsequent four entries reflect Declaration Numbers 1, 2, 3, and 4 made in the ROM.

The expression stack (the third record of the load module) contains two entries. The loader reads the first entry as follows: 1) Add the constant 14 to the expression accumulator; 2) Get the value word of that REF/DEF entry which begins at Word E of the REF/DEF Stack (a control section); 3) Change the value word, if necessary, to byte resolution and add it to the expression accumulator; 4) Store the result in that REF/DEF entry which begins 3 words into the stack (the DEF). The "result" signifies both the sum in expression accumulator, which goes into the value word of the DEF, and the resolution of the expression which goes into the resolution word of the DEF. Notice that a similar expression appears in a load item of the ROM, and that the loader built its expression entry by re-formatting the ROM's expression.

Looking at the second expression, the fact that Bit 9 of its first word is set indicates that this is a core expression. The expression says to add the value of that REF/DEF entry

beginning at word 7 of the Stack (the PREF), at word resolution, to a word in the core image. (In fact, the core image word is Word 0 of the fifth load module record.) This expression was constructed because the Loader could not completely satisfy the first "load relocable" load item in the ROM (which involves a PREF in the address field).

4.0 DESCRIPTION OF THE FIRST PASS

Overall execution of the Loader is controlled by the driver within the LDR segment beginning at location LOADER. Exit from the Loader back to CCI or PASS3 always occurs at location LEAVE within the driver. If an error occurs during processing, control is transferred to MESSAGE with the error number. MESSAGE builds the key, reads the ERRMSG file, prints the offending error (and the key) and transfers to LEAVE.

4.1 INIT1-INITIALIZATION FOR THE FIRST PASS

IN1 obtains memory by the method described in Section 2.4. It then zeroes its own data page (in LDR) and reads the LOCCT, ROM, and Tree Tables. Knowing the size of these tables, the declaration, REF/DEF, and expression stack pointers are now initialized. Sixty-four words are set aside for the declaration stack. The REF/DEF Stack follows. TOPOMEM is computed (from J:EUP in the JIT if UTS or on the basis of the number of pages given to the Loader if BPM), and the expression stack pointers are set.

Dynamic PLISTS are moved into dedicated areas of the DATA page for future use and, since CCI did not clear the last six words of each Tree Table, INIT1 does so now.

The ASSIGN record is scanned for F:number DCB names and these are entered as PREFs in the REF/DEF stack for future building by the Loader (if they do not get satisfied during PASS1). Unless NOTCB was specified, M:DO is also primary-referenced to allow for SNAPs and PMDs. If the load module is overlaid, M:SEGLD is primary-referenced for use by the segment loader. If BREF was specified, the library routine S:OVRLY is also primary-referenced. The load module file is opened and the information in the LOCCT

is moved into the OPENLM PLIST. In UTS, if the first word of the EXPIRE field is zero, the number of significant words in the EXPIRE control word of the OPEN VLP is set to zero. The system library is opened to prevent the alteration of the library while the Loader is using it.

If M:EF was assigned to labeled tape, the M:EF DCB has a 2 in the ASN field. All ROMs in the ROM Tables are then assumed to be on the labeled tape and are flagged by a 1 in bit position 30 in the third word of each ROM name. Load modules added from libraries are recognized as coming from disk, not tape, by not having this bit set.

For BPM, if M:LM has been assigned to a private volume, the (PERM, LIB) bit in the LOCCT is checked; the Loader will abort at this point if it is set.

Finally, the known sizes not associated with CSECTs or DSECTs are added to the TREE. These include the TREE size and the TCB size in 01 and 00 of the root. (For BPM, an obsolete feature is unfortunately still retained for compatibility - two words at the beginning of the root's 01 area are reserved and never used.)

The relationship of the LOCCT to the Tree Tables and ROM Tables are shown in Figure 14 and the linking among the Tree Tables is shown in Figure 13.

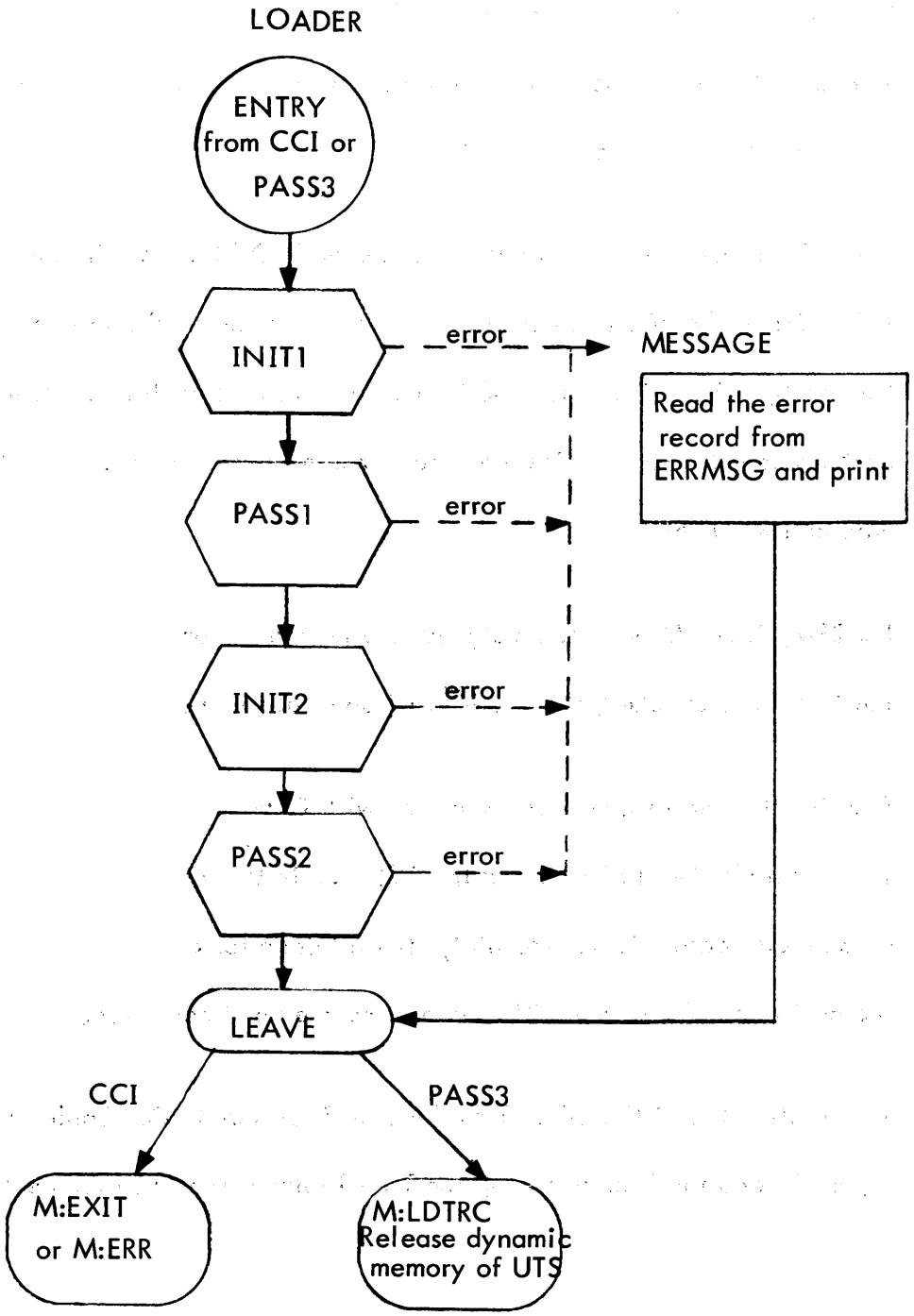


Figure 17. The Loader Driver (in LDR) Flow Chart

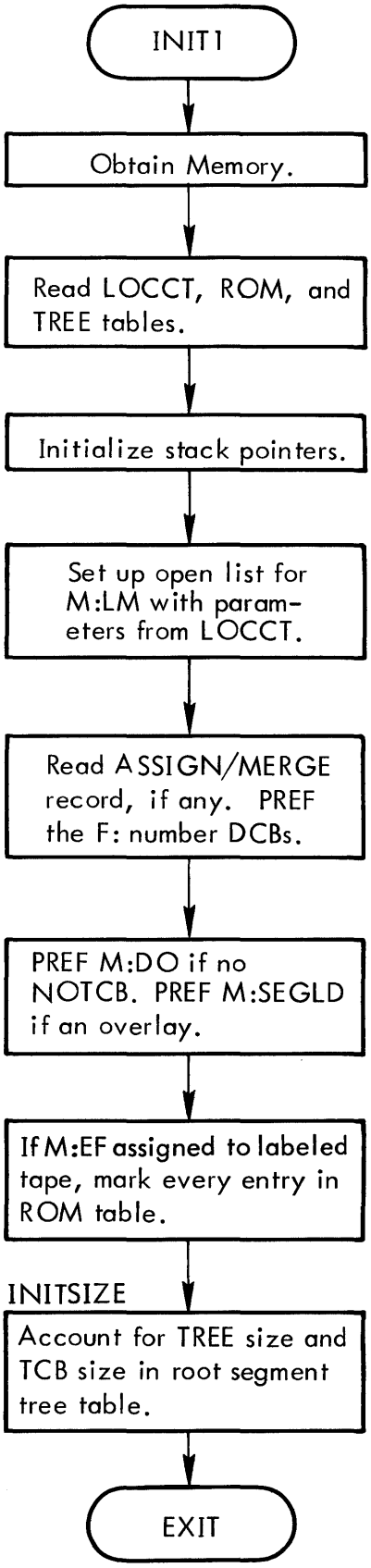


Figure 18. INIT1 Flow Chart

4.2 PASS1

We can think of PASS1 as consisting of four major parts: the main loop, the object module decoder (LP1), the load module processor (ADLDMD), and the librarian (SATREF).

4.2.1 The Main Loop

Starting with the root segment and proceeding along a path, the HEAD record of each input file named in the ROM Table for this segment is read and control directed to ADLDMD or LP1, depending upon whether the file is keyed or not (ROMs are sequential, load modules are keyed). At segment end, SATREF is called to augment the ROM Tables by library module names needed to satisfy PREFs (except PREFs to M: or F: DCBs). When there are no more forward links, PASS1 writes the current segment's stacks on the RAD, updates SEVLEV if there are PREFs (other than M: or F: names), and proceeds to the overlay links, then to the back links. See Figure 2 for processing sequence. When all of the segments have been processed and their stacks written, we will be sitting at the root segment. (Its REF/DEF stack is not written since PASS2 needs it immediately anyway).

At this point, all references to DCBs have been forced to the root. The root's REF/DEF stack is scanned for PREF DCBs and they are marked as Type B. FCOUNT contains the number of words needed for the DCB Name Table and is also accumulated during the DCB scan. In UTS, REFs to M:XX and M:UC ("special" DCBs not to be built by the

Loader) are satisfied from the corresponding values in the JIT. The entry is changed to a library DEF. Figure 3 illustrates the flow of the main loop.

4.2.2 Object Module Processor (LP1-Pass One)

All names (DEF, PREF, SREF) and control sections are "declared" by the ROM. Reference

to these items is by declaration number. This requires that the Loader associate a "declaration number" with every name and every control section.

Inherently, this number is a position in the declaration stack, every entry being a pointer to the entry in the REF/DEF stack which contains either the name or the protection type and size (if a control section). See Figure 19.

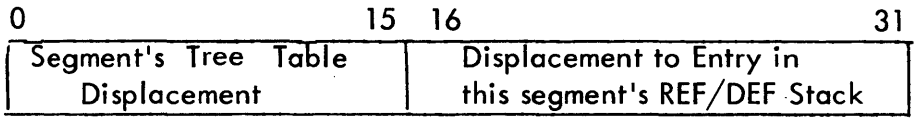


Figure 19. Declaration Stack Format

LP1 looks at all declarations (control sections and names) and all definitions (DEFs and forward references). It ignores all other load items. Every declaration results in creating an entry in the REF/DEF stack and an entry in the declaration stack which points to it. Every definition results in creating an entry in the expression stack whose destination is the REF/DEF stack entry which is being defined. The REF/DEF stack may gain one or more entries as a result of a definition whose defining expression involves an unknown forward reference.

a. Declarations - Declarations identify either control sections or names.

1. Control Sections

As control sections are encountered, the size is added to the appropriate protection type and in the segment's Tree Table for use by INIT2 in allocating buffers.

Declaration number 0 is special, being dedicated to a standard control section (DCSO) for use by one-pass compilers and assemblers. The Loader initially generates this declaration for expression reference; the processor will declare its size and protection type at the end of the compilation when it finally has this information.

2. Names

When a name is declared, LP1 makes an entry in the DECL stack. The name may have been previously entered in the REF/DEF stack via an object module or may now be added to the segment's stack. The appropriate type entry, i.e. DEF, PREF, or SREF, is added to the REF/DEF stack if the name is not found. In either case, the declaration will point to the segment in whose REF/DEF stack the name is stored and will indicate the relative position within that REF/DEF stack. A later module may change a PREF or an SREF to a DEF.

The routine which searches for names and adds them if necessary is ENNAM. Incidentally, all names beginning with M:, F: or F4:COM are forced to the root segment 's REF/DEF stack.

NOTE: A dummy section falls into both of the above categories. (See Section 2.1.1b.) Names that have been declared as DEF names may be redeclared as dummy sections, with the object language indicating size and protection type. Given dummy sections with the same name in different ROMs, LP1 will determine the maximum of the section sizes and accumulate it in the appropriate protection type and segment in the Tree Tables.

b. Definitions

Eventually, the ROM will define a DEF or a forward reference. That is, it will present an expression in terms of other declaration numbers (other names, control sections or forward references) which, when evaluated in the second pass, will yield the definition or VALUE (in the REF/DEF entry). For now, the Loader simply decodes the expression (in EXPRIN) and builds an entry in the expression stack whose DESTINATION is that entry in the REF/DEF stack indicated by the declaration number of the DEF or forward reference number. Declarations involved in the expression are converted to their REF/DEF pointers (picked up from the declaration stack entry) and stored in the appropriate WORD of the expression entry. If a Define Forward Reference and Hold expression mentions a forward reference number (add FREF), bit 10 of the corresponding REF/DEF entry is set for use by SQZ in WRITESEG.

References to FREF numbers that are not known cause these to be added to the REF/DEF stack. These FREFs will later be defined similar to DEFs (see Terminology, Section 2.1.1d.).

At module end, the forward reference numbers are released, severity level is accumulated, and control returns to the main loop of PASS1. Figure 20 illustrates the flow of LP1.

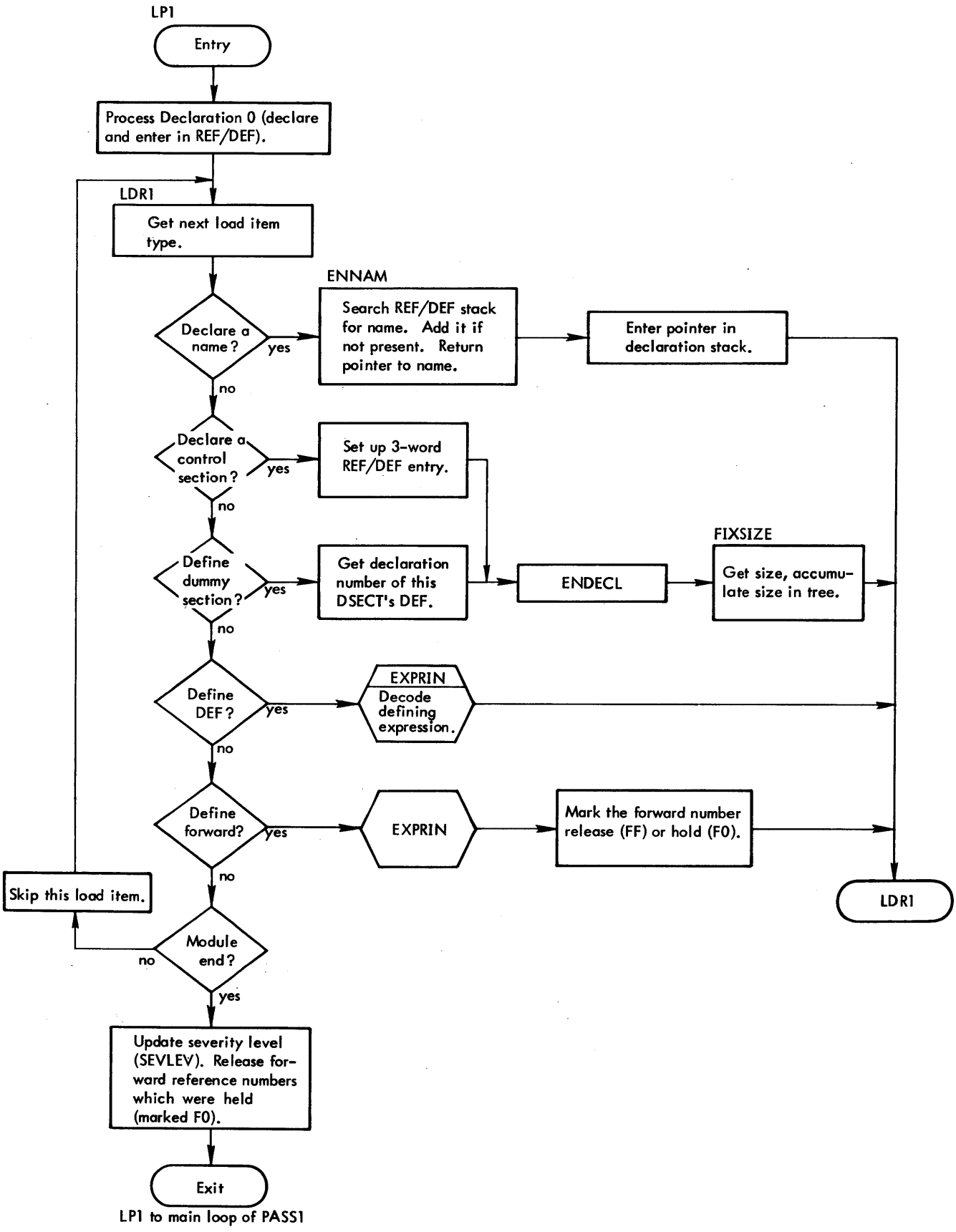


Figure 20a. PASS1 Object Module Processor (LP1) Flow Chart

EXPRIN - Expression Decoding Routine (in PS1)

Purpose: To decode a ROM expression which defines a DEF or forward reference and place a corresponding expression in the expression stack.

Input: (R7) = pointer to REF/DEF entry which is to become the destination of this expression.

(D2) = Declaration Stack Base

(D3) = Tree Table Pointer

(SR4) = return address

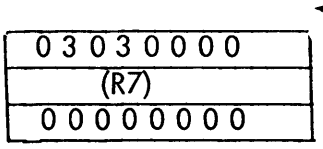
(SR2) $\neq$ 0 if expression is to be skipped.

Output: A new entry in the expression stack consisting of decoded expression.

The destination is from R7, and resolution = 0.

Comment: Expressions are decoded if they follow a Define DEF, Define Forward Reference or Define Forward Reference and Hold. Hence, this routine is entered for the purpose of decoding only from those three points in LP1. All other expressions are skipped in PASS1. The expression skip mode is determined by SR2 (SR2 $\neq$ 0 means skip.)

Flow: A skeletal entry is appended to the expression stack with resolution set to 0 and destination set with R7.



← Top of expression stack.

An expression control byte is gotten (CB_i) and inserted into its slot and the appropriate decoding routine is entered.

The decoding routine gets the item (constant, forward reference number, declaration entry, etc) which is to be stored in $WORD_i$ and branches to PTWRD which puts it in the new expression entry. If a forward reference is mentioned in the expression (add FREF) and this forward reference is new, it is added to the REF/DEF stack.

When the expression end control byte (02) is encountered, EXPREND1 updates the expression stack pointer, adds the size of the entry to the TREE and exits.

4.2.3 Load Module Processor (ADLDMD - PASS ONE)

A load module may be encountered as a result of either an EF specification or satisfying a PREF from a library. In either case, ADLDMD has at its disposal a header, a TREE, a REF/DEF stack, an expression stack, and the core image and relocation dictionary for one (and only one) protection type.

Using the space just above the REF/DEF stack, PASS1 reads the TREE record to determine

the REF/DEF stack size. The expression stack is read in just below the current expression stack and inverted, since the stack is being built upside down. All the expressions must be marked as unevaluated (bit 8=0) so that PASS2 logic can recognize the expression as such.

All core expressions in the load module (e.g., an expression that defines the address of an instruction in terms of an unsatisfied reference) have their destinations changed to be relative to the base of the load module. These will later (in EVL) have the control section base added to yield the correct destination word.

The REF/DEF stack is read in below the expression stack. An additional control section is added at the start which reflects the size of the entire load module (potentially many control sections) (See Sec. 3.2.3) The other control sections will be type 6 instead of type 4 and will hence be ignored by PASS2. The special control section also contains as the third word (normally, resolution) the relative position (within the expression stack being built) and size of the load module's expression stack so that the core expressions can be located and evaluated.

Each entry in the load module's REF/DEF stack is merged into the large REF/DEF stack. Control sections are added, and all named entries (PREFs, SREFs, and DEFs) are passed through CHKRFDF and are either added or not added according to whether the name had previously been encountered.

Forward REFs are flagged as "used" so that they will be ignored. Dummy sections are flagged as "defined". Space will be allocated for the entire module; reallocation of any

individual dummy section within the module is undesirable.

If the Loader generated the load module (as distinct from PASS2 of SYSGEN which also generates load modules), each entry in the REF/DEF stack has, as its value, the header of a chain (through the expression stack) of all words that pointed to that REF/DEF entry. The values are relative positions within the expression stack.

These values are replaced by the actual location of the REF/DEF entry. If PASS2 of SYSGEN generated the load module (indicated by the header, 81 being SYSGEN's PASS2 and 82 being the Loader), then each expression must be decoded control byte by control byte to find out which words are pointers to the REF/DEF stack. These are changed as above.

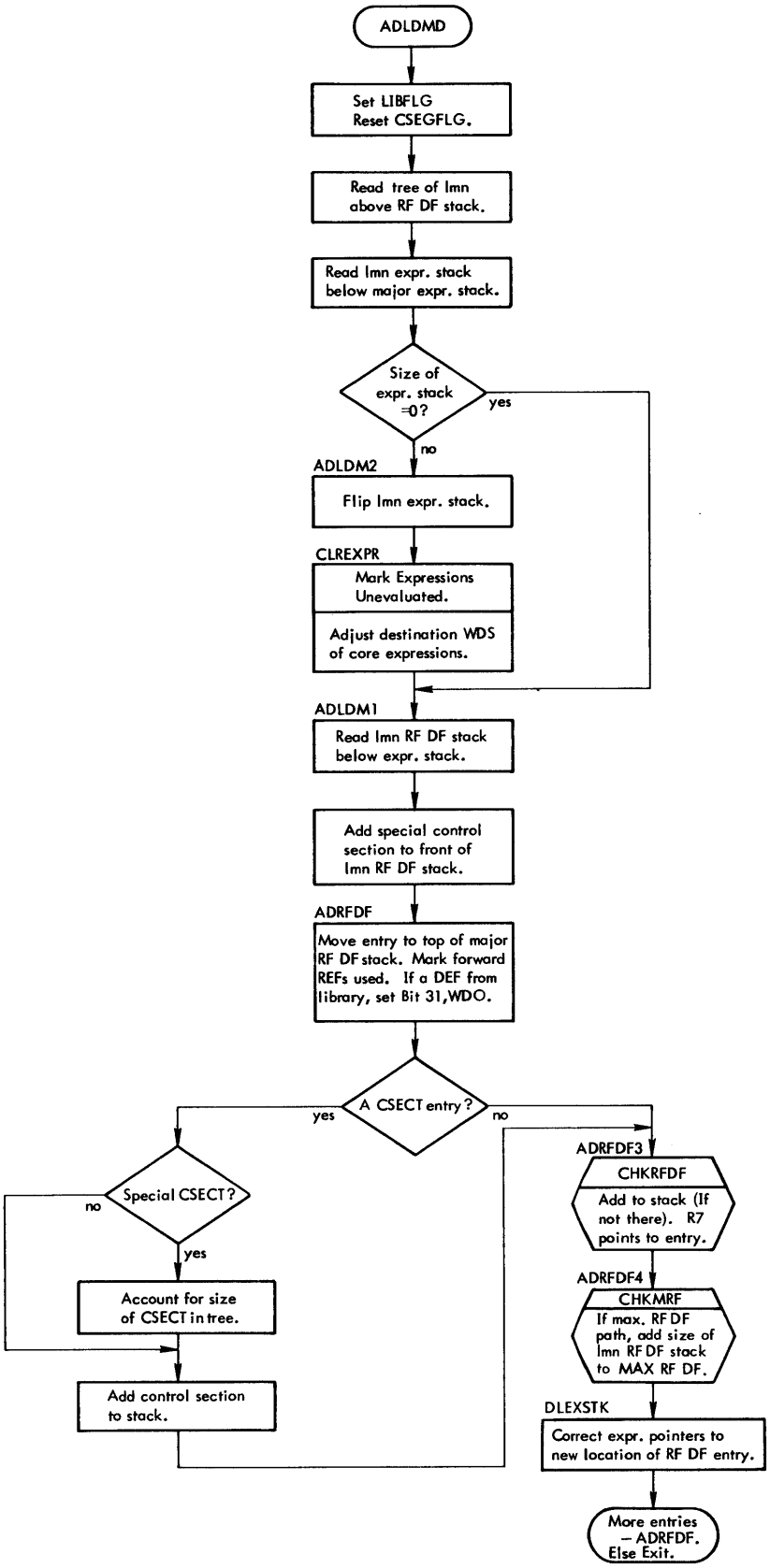


Figure 20b. PASS1 Load Module Processor (ADLDMD) Flow Chart

4.2.4 The Librarian (SATREF)

a. Load Module Libraries

The satisfy-reference logic works as follows: after each segment's element files have been read and its REF/DEF stack built, SATREF is called to satisfy the segment's PREF's by searching the specified libraries. Thus we attempt to satisfy all the PREF's we can in a lower segment before starting to build the REF/DEF stack for a higher segment.

The purpose of this approach is to handle the situation where a high segment contains a PREF which is also contained in a lower segment and the corresponding DEF is in a library. It is certainly desirable to have the library routine containing the DEF in the lower segment (otherwise the high segment and all of its backward path would have to be in core every time the lower segment needs this DEF). Note that this method produces the following result: if a low segment has a PREF whose corresponding DEF is located in both a higher segment and one of the specified libraries, the library DEF will be used.

SATREF initiates the library search by checking the LOCCT for UNSAT account numbers. The first dictionary (:DIC) is opened and the segment's REF/DEF stack is searched for the lowest (alphanumerically) PREF. This name is used as the key for reading the dictionary. If the response is "no such key," the alphanumeric search continues through the stack for the next lowest PREF. If the read is successful, the record read contains the name of the load module with the DEF corresponding to the PREF key, and the load module is merged with the other input files in the manner described below.

In either case, the search continues until all of the segment's PREFs have been checked against this dictionary. Then the first dictionary is closed and the next dictionary is opened.

Each time a library load module is to be merged with other input files, room is made for inserting an entry in the ROM table at the end of the entries for the given segment. The last ROM bit is set on the previous entry, and reset on this entry. It is also flagged as coming from a library to save unnecessary opens and closes later. (See Figure 8.) All other tables are moved up eight words in memory to make room for the insertion (the extra word maintains even-word boundaries on the REF/DEF stacks). Pointers from the TREE to higher parts of the ROM Table are adjusted up by eight words.

The name is transferred to the ROM Table and to the open element file PLIST. If not already open, :LIB is opened. The header is read into BUF with the key LMN concatenated with HEAD. Control goes to CHECKROM which verifies the header and calls the load module processor ADLDMD to form the appropriate stack entries. The routine then returns for the next PREF.

When there are no more PREFs and no more accounts, control returns to the main loop of PASS1.

b. Core Libraries (UTS only)

The association of core library is triggered by one of two conditions:

a. A PREF to 9INITIAL (FORTRAN) or 9DBINIT (FORTRAN DEBUG)

b. The presence of a :Pn in the UNSAT list on the !LOAD card.

In ENNAM, a record is kept in word CORELIB if 9INITIAL or 9DBINIT is encountered

as a PREF.

In the SATREF loop, CORELIB is checked as is the UNSAT list (for a :Pn). If either condition dictates a core library, the :Pn HEAD is read to determine the core library's context size. This is retained in CORELIB for future use by ALLOCATE in PASS2, which must bump the DATA location counter (DLOC) accordingly. Control is transferred to ADLDMD (via CHECKROM) in order to merge the DEFs of :Pn in with the REF/DEF stack.

The association of core library is inhibited if (PERM, LIB) is specified or if the load module name begins with the characters :P. This is done by setting CORELIB to -1 in IN1.

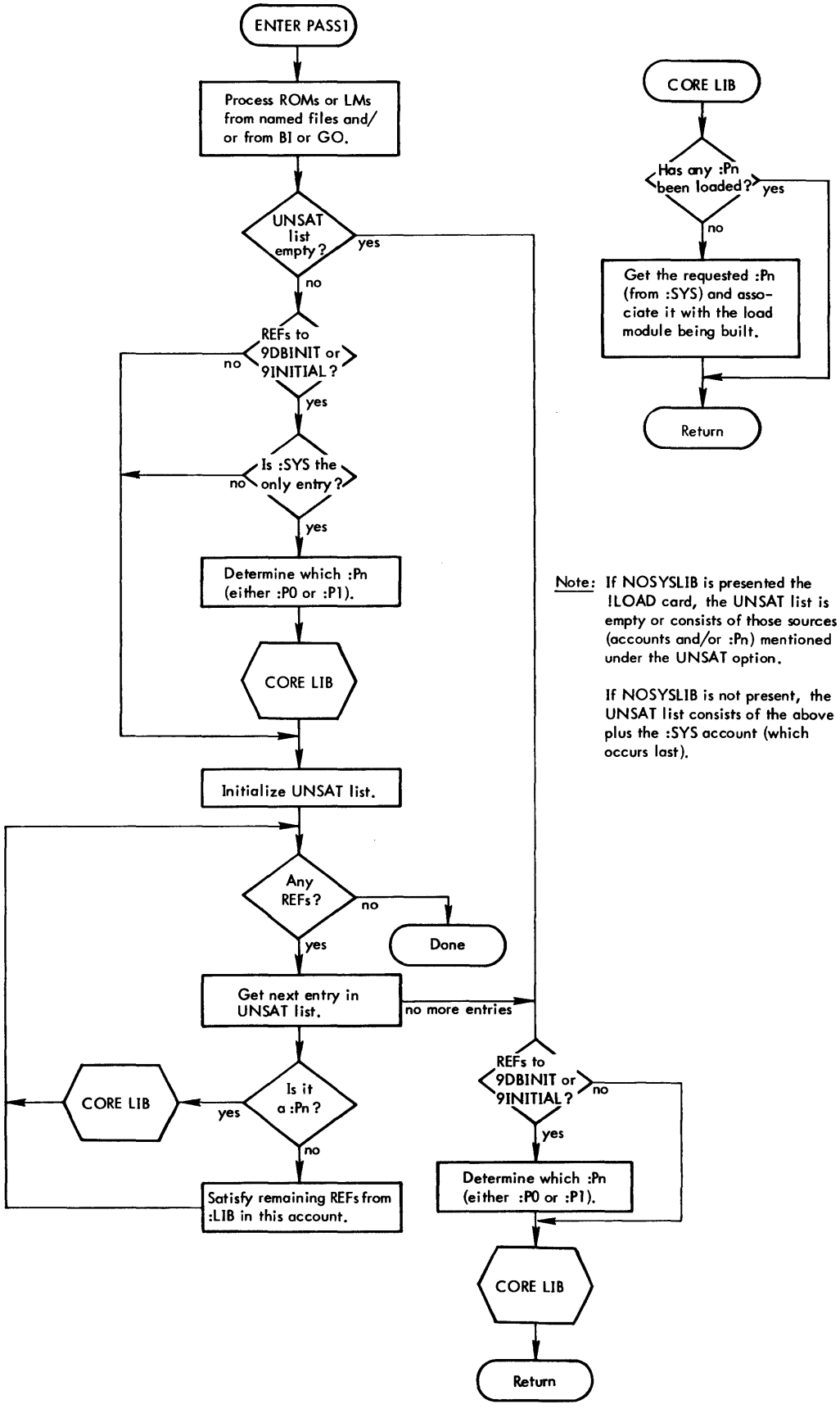


Figure 21. Core Library Association Flow Chart

5.0 PREPARING TO FORM THE CORE IMAGE

5.1 IN2

INIT2 contains the logic which partitions memory for PASS2 usage. It also determines the size of each protection type area for the final load module. First, the size of the TCB and library error tables is accounted for, the necessary information being in the LOCCT. The DCB Name Table size is calculated from FCOUNT which was computed at the end of PASS1 (two is added for the top and bottom of the table).

Then each path of the TREE is followed, and the sums of 00, 01, and 10 segment sizes are accumulated in D1, D2, and D3. When a segment has no sublink, these sums are compared with SR1, SR2, and SR3, respectively to determine the maximum path for each protection type. Also, the large protection type for a single overlay segment is retained in MAX00, 01 and 10. This is done in order to allow for CSEG buffers of the maximum size.

ALLMEM is called once to allocate buffers for the core image and relocation dictionary (unless absolute) of the root segment, and again with the values MAX00, 01 and 10 for current segment loading. The double buffering permits dummy sections in the root and higher segments all to store into the section in the root. The byte addresses of these buffers are in RSEG00 through CREL10. The buffers are allocated from the top of memory (TOPOMEM), down.

The load module's location counters are held in DLOC, PLOC, and SLOC (00, 01 and 10 respectively). They initially represent the beginning of each of the three TREES. The bias or background lower limit is used as the beginning value of DLOC, and the

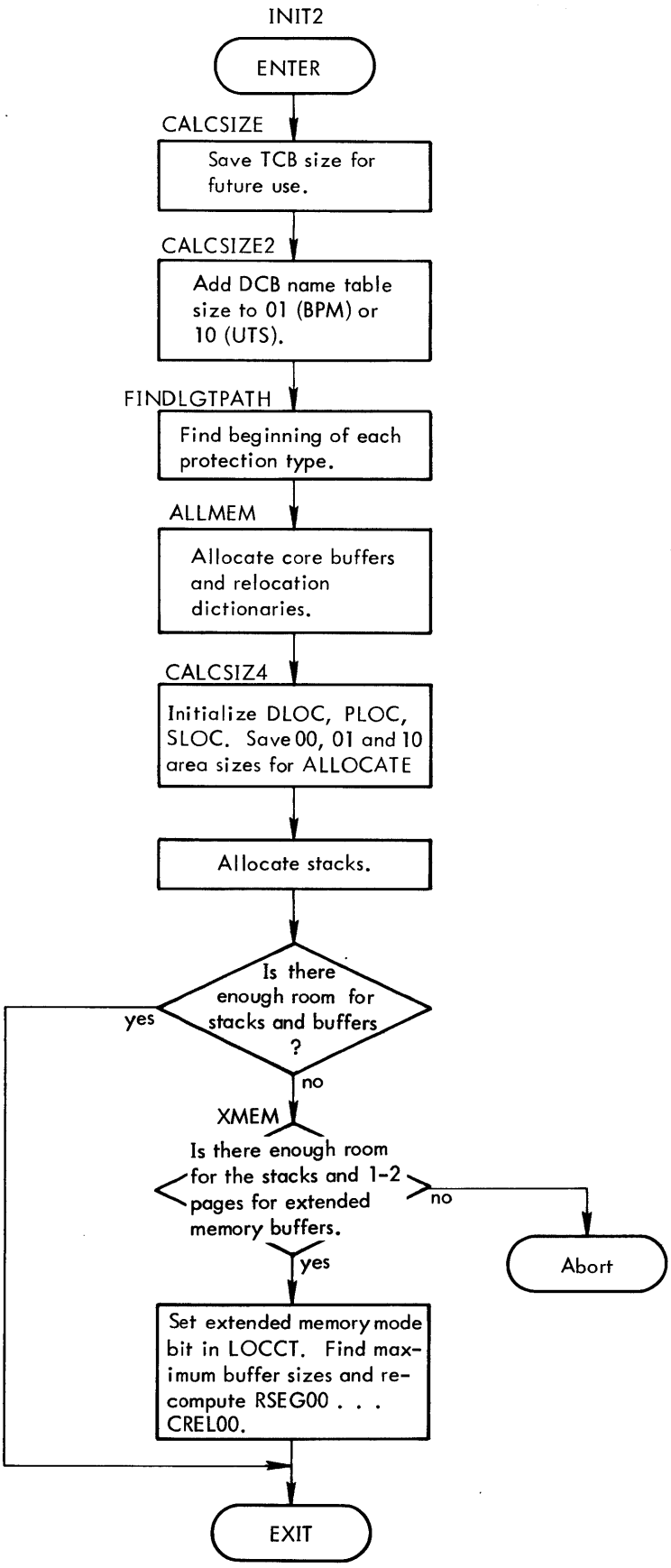


Figure 22. INIT2 Flow Chart

PLOC and SLOC are computed. These values plus the total sizes of the 00, 01, and 10 areas, respectfully, are saved in BUF-BUF+5 for generation of the allocation summary by ALLOCATE.

We now have to allow space for the maximum stack paths.

In PASS1, the maximum REF/DEF and expression stack size was saved. It is known that the REF/DEF stack will not grow and also that the declaration stack is still at its maximum size. The expression stack is allocated immediately above the REF/DEF stack, and the top of it is compared to the bottom of the buffers. If there is enough room, PASS2 begins with memory partitioned as shown in Figure 8; otherwise we determine whether extended memory mode can be entered. If so, the maximums of

$$C\begin{Bmatrix}SEG\\REL\end{Bmatrix}\begin{matrix}00\\01\\10\end{matrix}, \quad R\begin{Bmatrix}SEG\\REL\end{Bmatrix}\begin{matrix}00\\01\\10\end{matrix}$$ are computed and

the buffer pointers for the current segment and the root are set equal. Hence for the concatenation phase of id there will be six (or three) buffers to work with. See Figure 9B.

5.2 PS2 - THE DRIVER FOR THE SECOND PASS

PS2 is really a driver for the second pass. It calls ALL, EVL, and WRT as it proceeds along the segments. Figure 5 illustrates the overall sequence for this pass.

5.3 ALL - MEMORY ALLOCATION

Refer to Figure 1 for memory layout of the load module being formed. DLOC, PLOC, and SLOC -- the three location counters for 00, 01, and 10 -- have been established

at their beginning values by INIT2. If the segment being allocated is the root segment, we save a pointer to the TREE and increment the PLOC location counter by the TREE size.

We save a pointer to the DCB Table and increment PLOC (or SLOC if UTS) by the DCB Table size. If REF or BREF was specified, we increment PLOC by the number specified by the user, or supply the default.

PLOC now has the location of the first control or dummy section. Control goes to LOADFO and LOADM to allocate the F: and M: DCBs (Still only for the root segment).

For UTS, if rounding has occurred to prevent DCBs from overlapping page boundaries and the adjustment did not fit in the RSEG10 buffer, it must be taken into account at this time by readjusting the Loader's buffers for protection type 10 (refer to Figure 8). The additional size is accounted for in the root's tree and, if the load module is relocatable, in the root's relocation dictionary. Buffers are moved down for the root's and current segment's core image buffers and for their corresponding relocation dictionary buffers if the load module is relocatable. If in nonextended memory mode the buffer shifts result in a collision with the expression stack, the Loader will abort at this point.

Next all 01 protection type sections are allocated by putting PLOC into the value word, setting the resolution, and adding the size to PLOC. Then we go to work on the 00 protection area, first accounting for Blank COMMON*, then establishing the TCB pointer, and then appropriately incrementing DLOC (root segment only). All 00 protection type control and dummy sections are then allocated. Finally, using SLOC, all 10 protection type sections are allocated.

A final run is made through the REF/DEF stack to put values in the control sections read from the library. Since these are all type 6 entries, they were not allocated; therefore, the value of the last type 4 entry is put in the first type 6 entry encountered; that section size is added and put in the next type 6 entry, and so on until a new type 4 entry is encountered.

If the segment just allocated is the root, the allocation summary is output, including a possible adjustment in the 10 size for UTS as a result of rounding DCBs to prevent overlap on a page boundary.

*If UTS, we first account for the core library's context area.

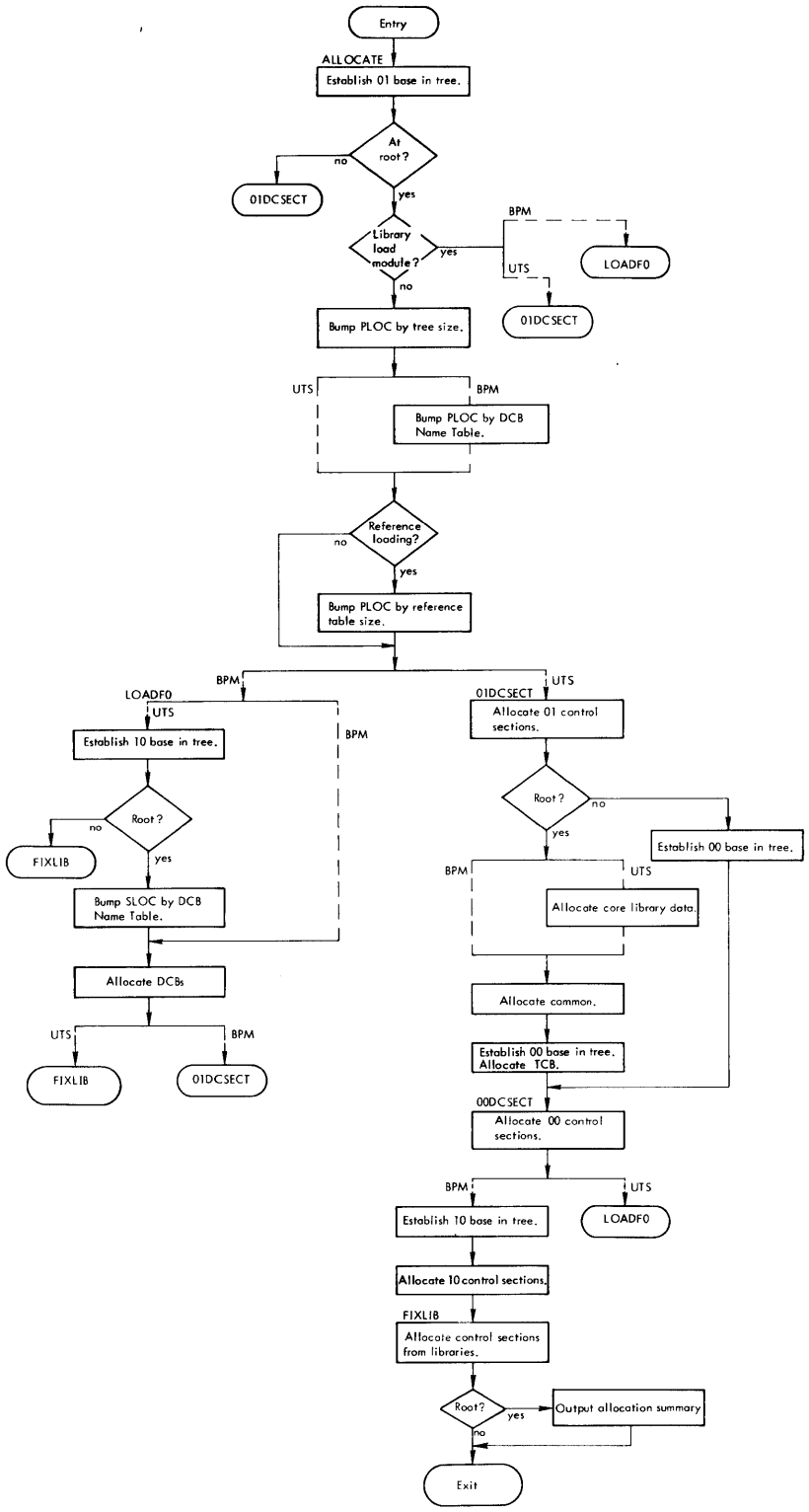


Figure 23. ALLOCATE Flow Chart

6.0 FORMING THE CORE IMAGE (EVL)

EVL is entered from PS2 once for each segment, beginning with the sublinks, to the overlay links, and back down toward the root (see Figure 4). It has two entry points, EVEXPRS and LOADSEG. PS2 first calls EVEXPRS to evaluate all expressions for this segment which were formed during the first pass. It then calls LOADSEG to actually form the core image and relocation dictionary by reprocessing the object language of a ROM or reading in and relocating the core image of a load module.

6.1 EVEXPRS

Since all control and dummy sections for a given segment have been allocated at this point, we are in a position to evaluate the expressions which are typically in terms of these values plus constants. Because some expressions will be in terms of other DEFs, every expression in the stack must be evaluated repeatedly until one complete pass has been made during which no expressions were evaluated. Evaluating an expression consists of decoding the expression's control bytes. (Note that we are "decoding" those expressions which are already in the expression stack from the first pass. Since we are not forming the stack entry, EVEXPRS is a much simpler version of expression evaluation than the EXPRIN routines found in PASS1 and LOADSEG.)

If the byte is either an add or subtract declaration or a forward reference, the corresponding entry in the REF/DEF stack is picked up if it is defined. If it is not defined, the expression cannot yet be evaluated. The other control bytes add constants or affect the resolution. When the expression is successfully defined, the value and resolution are put into the REF/DEF entry pointed to by the destination word of the expression. Core expressions

(which come from load module expression stacks in PASS1) are ignored at this point. (This routine may also be entered (later) from ADLDMD for the purpose of evaluating core expressions which come from load modules.)

6.2 LOADSEG

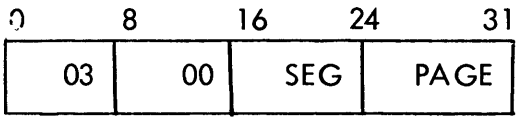
LOADSEG can be viewed as consisting of three major parts: the main loop, the object module processor (LP1) and the load module processor (ADLDMD). (Notice the similarity between LOADSEG and PASS1.)

6.2.1 The Main Loop

The main loop begins by initializing the relocation dictionary buffers if XMEM is inot in effect. The buffer is filled with E's or 0's for the current or root segment, respectively. The segment name is printed at top-of-form if a map was requested and a !TREE card was present. LOADSEG now begins to reprocess the input files by running through this segment's ROM table. Control is directed to LP1 if the module is a ROM or to ADLDMD if it is a load module.

Both LP1 and ADLDMD are concerned with developing the core image. The logic of extended memory mode (XMEM) will come into play for every word of the core image and every relocation digit which is constructed. When information is about to be stored into a buffer (core or relocation) and extended memory mode is in effect, a three-byte key is created consisting of the segment number and a page number. (For a standard load module, this number corresponds to the page address of the buffer this record will go into during the concatenation process. For the paged load module, this number corresponds to the page containing this record at execution time.) The key is compared to the key of the page currently in memory. (Recall that there are only one or two buffer

pages at TOPOMEM.) If the keys are not the same, the page in memory is written out and the new key is used to read in the desired page.



SEG = Displacement within TREE Table of this segment's entry

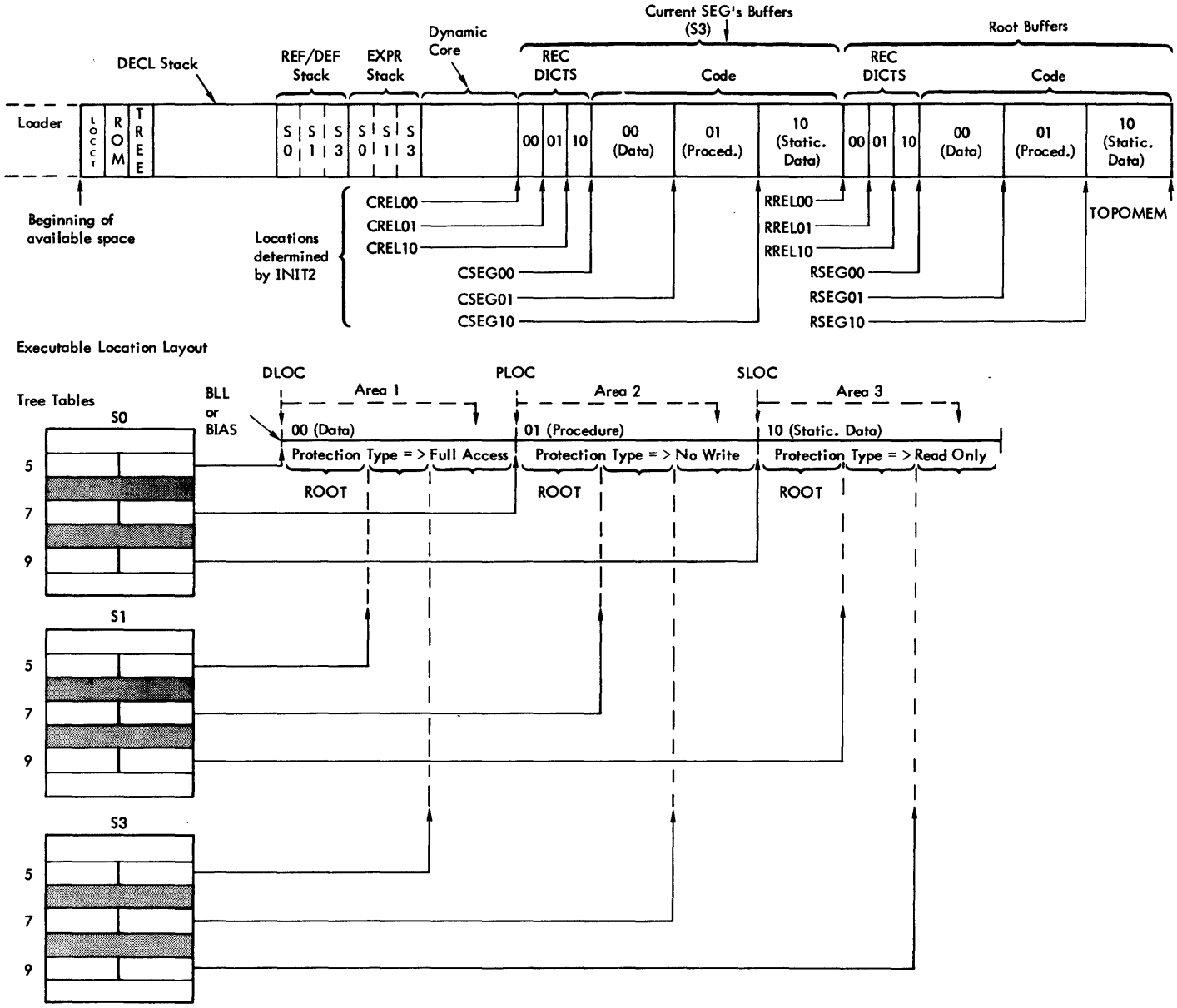
PAGE = Page number of the concatenation buffer.

Figure 24. Format of the Keys of idX (Extended Memory File for Standard Load Module)

6.2.2 Object Module Processor (LP1-PASS TWO)

An object module is processed through straightforward decoding of the load items. The main loop of LP1 is at LDR1 which contains a jump table to the individual routines. A control byte of Module End terminates LP1 and control returns to the main loop of LOADSEG (at NEXTROM). The load items fall into two categories: those which were handled in Pass One (declarations and definitions) and those which were ignored in Pass One (start address, origins and items which result in words or bytes in core).

Of the first category, LP1 handles declarations and definitions as follows: A declaration stack is formed again so that expressions can be related to their REF/DEF components. Name declarations are handled by looking up the name in the REF/DEF stack, forming a pointer to it and entering the pointer in the declaration stack.



Conditions:
1. BPM.
2. Nonextended Memory Mode.
3. CSEG = S3 (see Figure 2).

Figure 25. Snapshot of Core Usage During EVL

Control sections are handled by looking through the REF/DEF stack for the first TYPE 4 entry. The type is then changed to a 6 to prevent its being used again, the pointer is formed and entered into the declaration stack. Expressions which define DEFs and forward references are skipped. Define forward marks the FREF entry in the REF/DEF stack with an F0 and FF (depending on Define Forward Reference or Define Forward Reference and Hold, respectively). Forwards with F0 are marked with FF at module end to prevent their being used again.

We now consider load items of the second category, and these are, of course, the heart of LP1.

Define Start is handled by evaluating the expression (EXPRIN), shifting the obtained value to word resolution and storing it in START (for later placement in the HEAD record).

LP1 switches from one control or dummy section to another by an origin. The ORIGIN control byte (from the ROM) is the only means by which the Loader determines where data is to be placed within a control section. (Note: It is the responsibility of the ROM to present at least one ORIGIN control byte for every control or dummy section.) The expression defining the origin is evaluated (it must be evaluatable and have resolution) and the value obtained is shifted to byte resolution. The value is then compared with the bounds of three protection types of the current segment and of the root segment. It must be within one of those segments. Once the appropriate segment and type are discovered, the base of that section is subtracted and the base of the corresponding buffer is added, yielding the appropriate byte address at which to place the next load item. This value is put into the location counter, LOC. The segment base and buffer base are remembered in BIAS and FBIAS, respectfully, for possible use by XMEM.

Basically, LP1 is concerned with load items that result in words or bytes in core (that is, from a Loader perspective, they result in words being placed in the segment buffers or the XMEM file). These items are Load Absolute, Field, Load Long Relocatable, and Load Short Relocatable (see Sigma Object Language). These items are either absolute or contain expressions involving the base of a control section, a forward reference or some combination of externals. The expression evaluator, EXPRIN, is used to decode and evaluate the expressions. Unless the load module has unsatisfied references, values are obtained and the load item is placed in the core image buffer. The relocation digit is calculated and placed in the relocation buffer.

If there is an unsatisfied reference where an instruction references external data, the absolute part of the instruction is put in the core image and a "core expression" is added to the expression stack.

Core expressions left in the expression stack in this manner are, in general, meaningful when the load module being formed is to become part of the library. In this case, the core expression would be evaluatable when the load module is combined with other ROMs since the PREFs would presumably have been satisfied. ADLDMD (which would be handling the module) would do the evaluating and would insert the value into the field part of the absolute instruction in the core image.

a. Load Absolute

The simplest item is Load Absolute. This load item contains a byte count followed by the number of bytes that are to be placed sequentially into the core image,

beginning at the current value of the location counter. The relocation digit for these absolute load items is X'E'.

b. Field

The field allows an expression to be evaluated and added to any width and any position in a word or words. Since this logic handles all relocatable items, it includes the development of the relocation digit.

Before the expression is read, the relocation digit is initialized. If the field terminates at the end of a word, the relocation digit will be 0, 1, 2, or 3, according to whether resolution is byte, halfword, word, or doubleword. If the field does not terminate at the end of a word, left-half doubleword resolution or both-halves doubleword resolution is checked for. If none of these criteria are met, then the item is absolute.

Next a core expression destination word is constructed (See Section 3.2.4). The expression is evaluated (EXPRIN) and, if it is not absolute, the relocation digit is calculated. If it is not evaluatable, FIELD exits. (At this point, a core expression has been added to the expression stack; a stack overflow may have been encountered if there was no room for the expression.)

For expressions that have resolution, the relocation digit is the resolution control (0 for byte, 1 for halfword, etc.), if the field is right-adjusted. Resolution control is the output from WHATRES in R2. In the case of doubleword resolutions in halfwords, the resolution digit already present is checked for the case where both

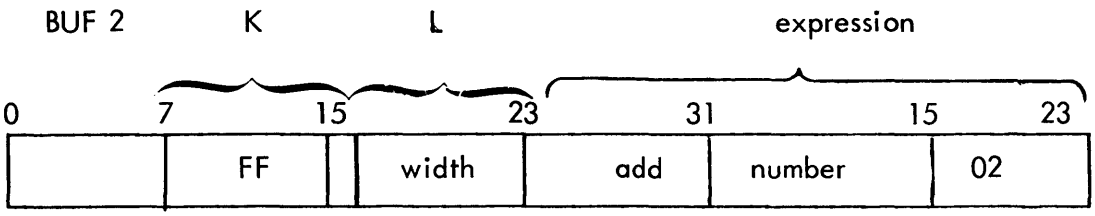
halves must be relocated.

Next, the destination word is used to add the value of the expression to the appropriate field in the core image. Remember that this field may extend backward across as many as 8 words.

Finally, if reference loading has been specified (REF or BREF), the CAL and PLIST must be constructed. During the expression evaluation, the highest segment above the current segment referenced in the expression was remembered in RFLDSG (0 means that no REF - loading is required). A pointer to the next available location for building the PLIST is kept in the last word of the root segment of the TREE Table. The PLIST is constructed in SR1 through SR4 with the call formed in SR3, and exchanged with the word in memory requiring REF-loading. The PLIST is put away in the area saved in the 01 root segment and the field logic finally exits.

c. Load Long/Short Relocatable

Both of these load items contain a four-byte word and a declaration or FREF number to be added to the word at a given resolution. (Short form assumes word resolution and a six-bit declaration number.) For these forms, a byte string that looks like a ROM field expression is created in BUF2. (See Define Field in Object Language, BPM Reference Manual.) It has the form:



where:

FF determines the location of the field.
Rightmost bit is location minus 1 bit.

width is 19 bits less specified resolution: 0 bit for byte; 1 bit for halfword; 2 bits for word; 3 bits for doubleword.

add is 20 bits for add declaration, 24 bits for add constant at the appropriate resolution.

number is the two-byte forward reference or the two-byte declaration number if there are over 100_{16} declarations;

or the one-byte declaration number followed by an 00 (padding) if there are fewer than 101_{16} declarations.

02 is expression end.

The four absolute bytes are then placed in memory at the location pointed to by the location counter that is incremented to the next word. (The location counter must begin at a word boundary or we have an ILLEGAL BOUND.) Certain pointers are then switched so that the field logic will get the expression from BUF2 rather than from the standard input buffer (BUF) and the FIELD logic is called.

Figure 26 illustrates the general flow of LP1. Two important subroutines of LP1 are EXPRIN and FIELD, illustrated in Figures 27a, 27b, and 27c.

LP1 Routines

DDNAM	(Declare DEF name) - Locate the name (LOCRFDF) and declare it (ENDECL).
DPNAM	(Declare REF name) - Locate the name (LOCRFDF) and declare it (ENDECL).
DSNAM	(Declare SREF name) - Locate the name (LOCRFDF) and declare it (ENDECL).
ORG (Origin)	Evaluate the expression which follows (EXPRIN). Shift to byte resolution and store value in RLOC. Determine which segment and protection type the ORG value is in, then compute the Loader's location counter, LOC (=ORG value - SEG base + buffer address.)
DFREFH or DFREF	Define forward - Locate entry in stack and mark it with F0 and FF. Skip defining expression.
DDSECT	Declaration # is fetched and DSECT declared.
DCS0, DCS	Locate next control section in stack. Change type from 4 to 6 and declare.
DSTART	Evaluate the expression which follows EXPRIN. Shift value to word resolution and save in START.
MODEND	Update severity level, release all forward REF numbers, exit LP1.
FIELD	Form the destination word stack. Evaluate the expression which follows (EXPRIN). If a value is obtained, calculate reloc. digit, store in buffer and store value in buffer. If no value, leave expression stack and exit.
LABS	Fetch bytes and place in buffer.
LSREL	Create a field type expression BUF2. Store the four-byte item in buffer.
LLREL	Call FIELD to evaluate the expression and store in buffers.

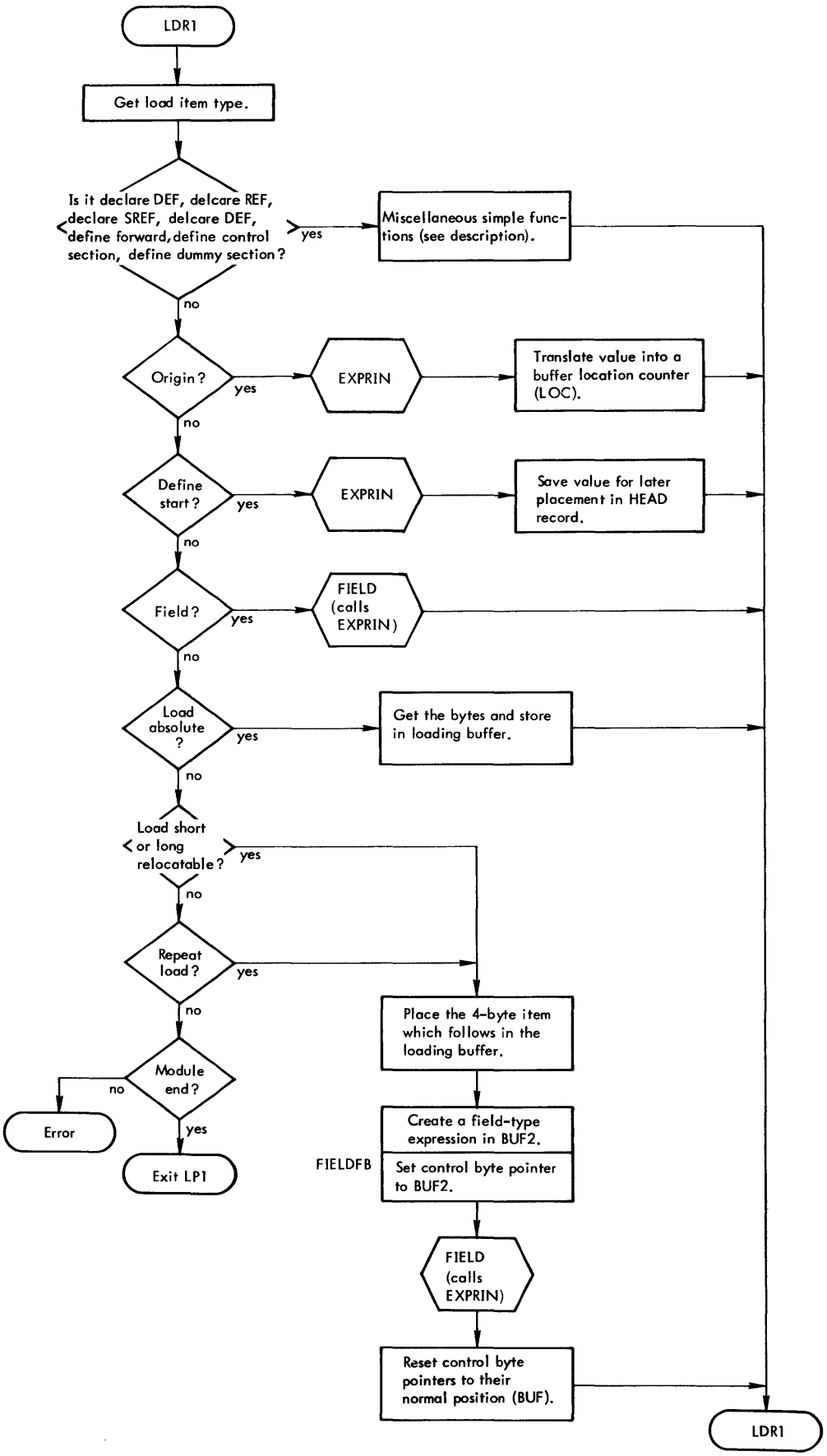


Figure 26. PASS2 Object Module Processor Flow Chart

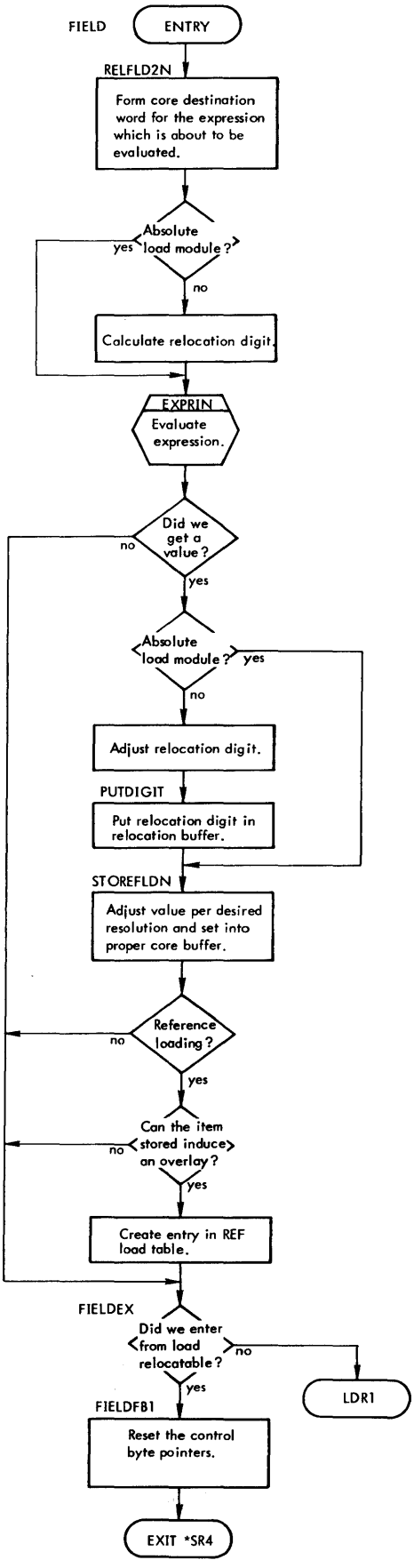
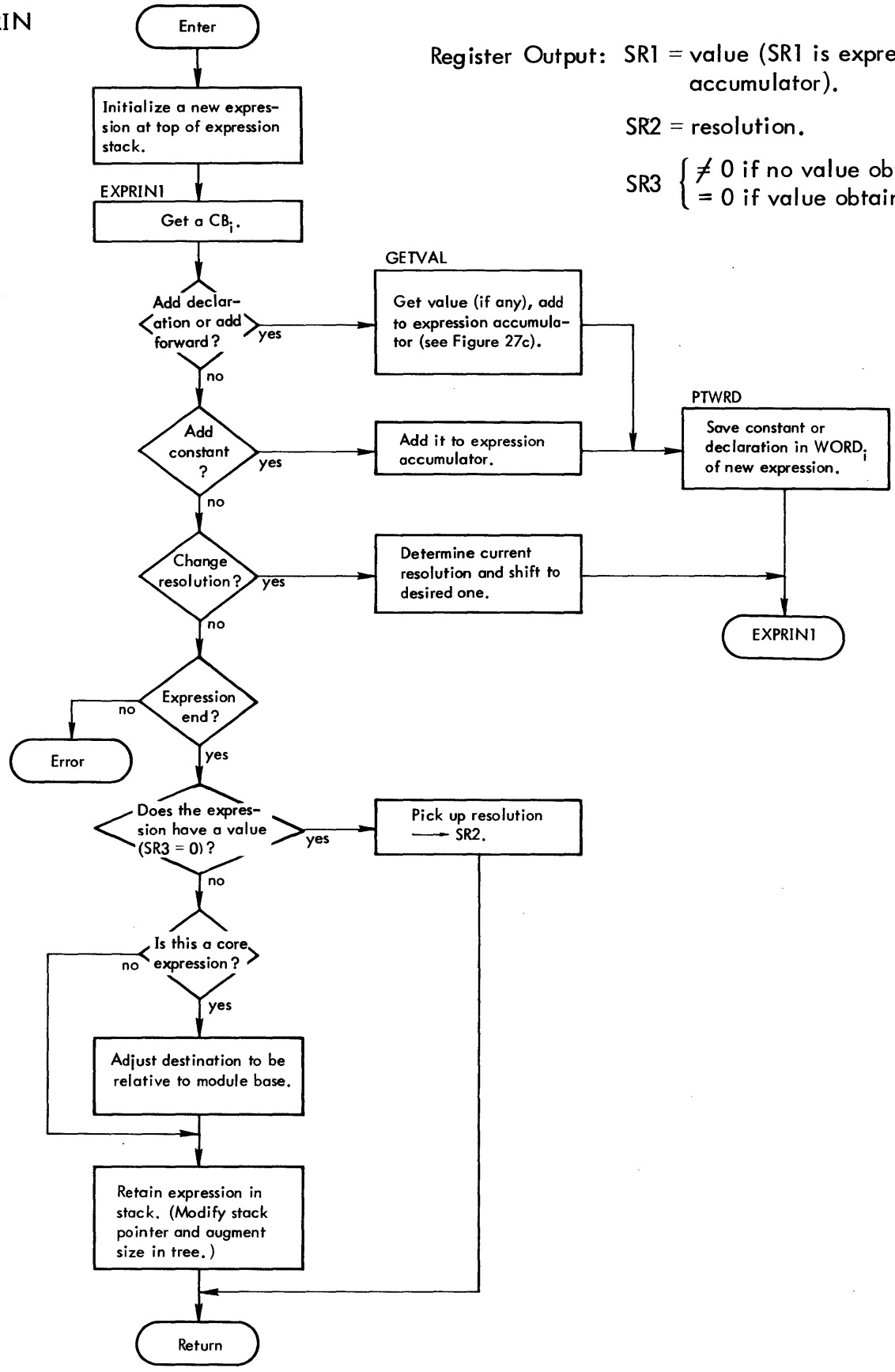


Figure 27a. Field and Expression Logic Flow Chart

EXPRIN



Register Output: SR1 = value (SR1 is expression accumulator).

SR2 = resolution.

SR3 { ≠ 0 if no value obtained. = 0 if value obtained.

Figure 27b. EXPRIN Flow Chart

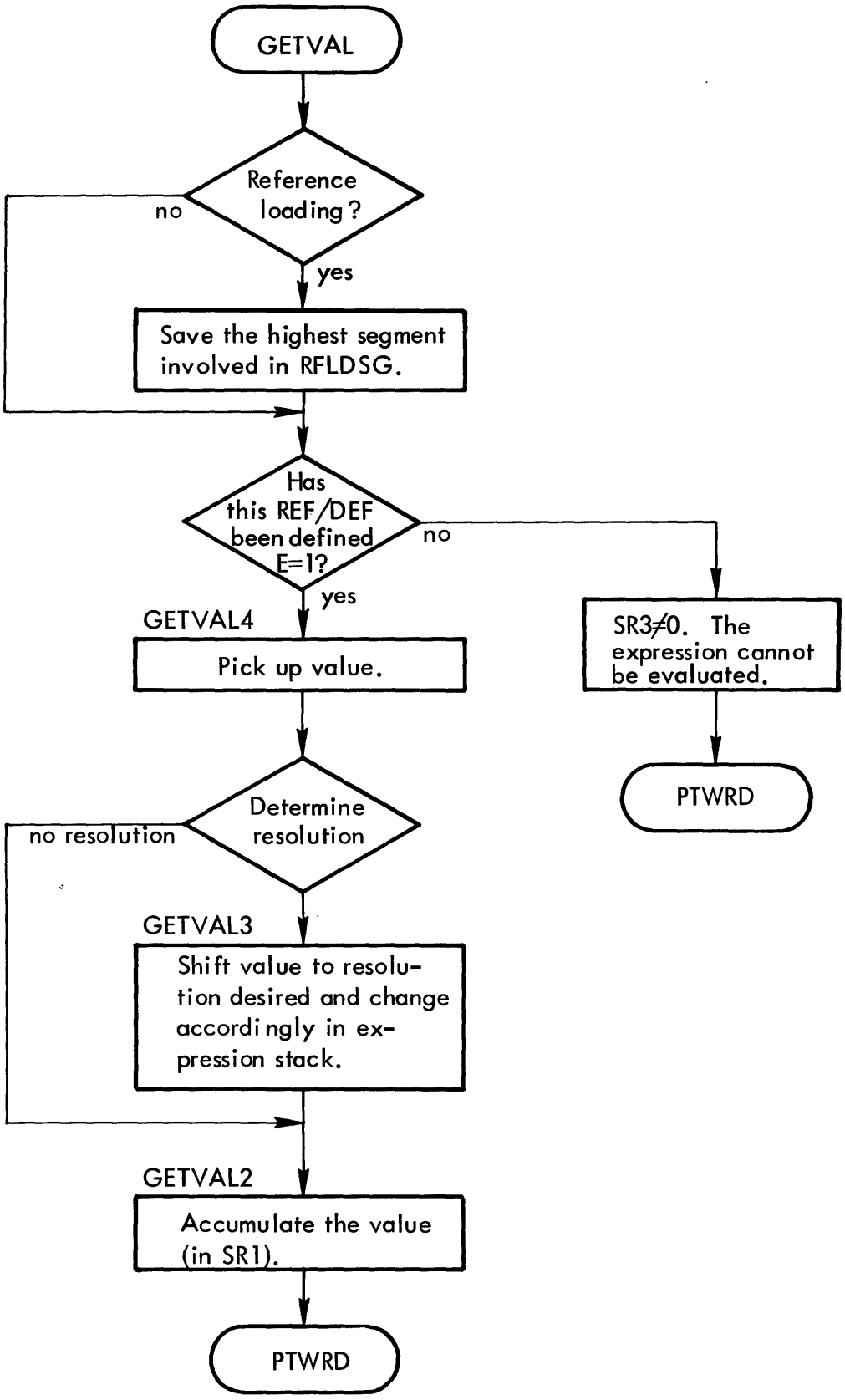


Figure 27c. GETVAL Flow Chart

6.2.. 3 Load Module Processor (ADLDMD - PASS TWO)

When a load module is encountered, ADLDMD is called. The special control section inserted in PASS1 in the REF/DEF stack containing the address and size of the module is located. The buffer address for the core image in the appropriate Loader buffer is calculated (in extended memory mode, the image is read in above the expression stack). Next, the relocation dictionary is read into its buffer if the mode is not ABS or extended memory. In ABS mode, the relocation dictionary is read in above the expression stack. In the extended memory (XMEM) mode, it is read in above the core image which is then relocated by interpreting each relocation digit and adding the appropriate bias to the corresponding word. Next, if we are in XMEM mode, each word of the relocated image is stored through the XMEM logic and the same is done for the relocation dictionary if the module is not ABS.

Since we know the relative beginning and size of the module's expression stack from the special control section, we can now evaluate the core expressions in the module's stack and resolve any words whose addresses were in terms of PREFs that are now satisfied by other modules. The evaluation is performed in the EVEXPRS section of EVL. The relocation digit for each word must also be corrected. The value and relocation digit for the core expression is then stored (through XMEM logic, if necessary).

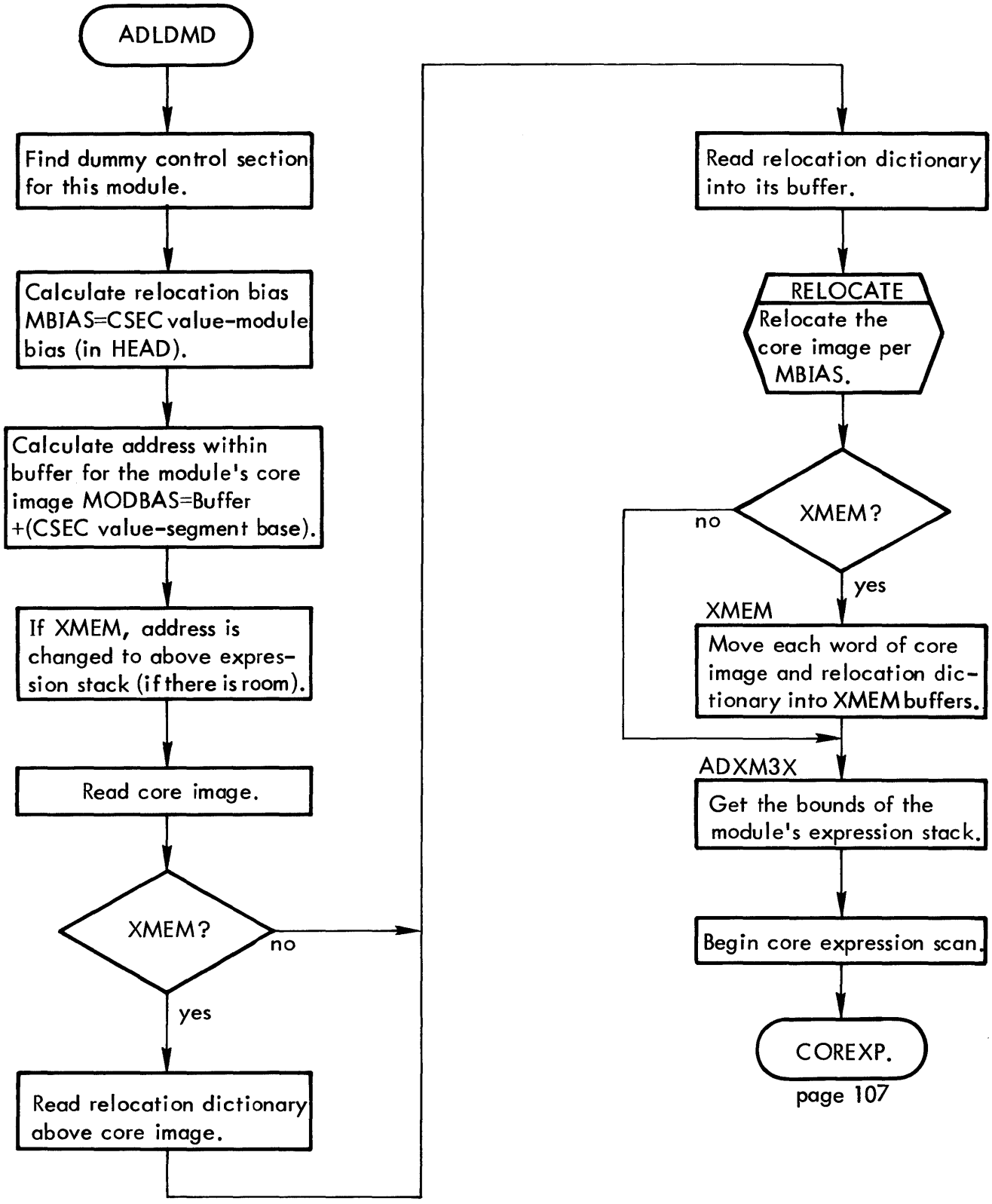


Figure 28. PASS2 Load Module Processor Flow Chart

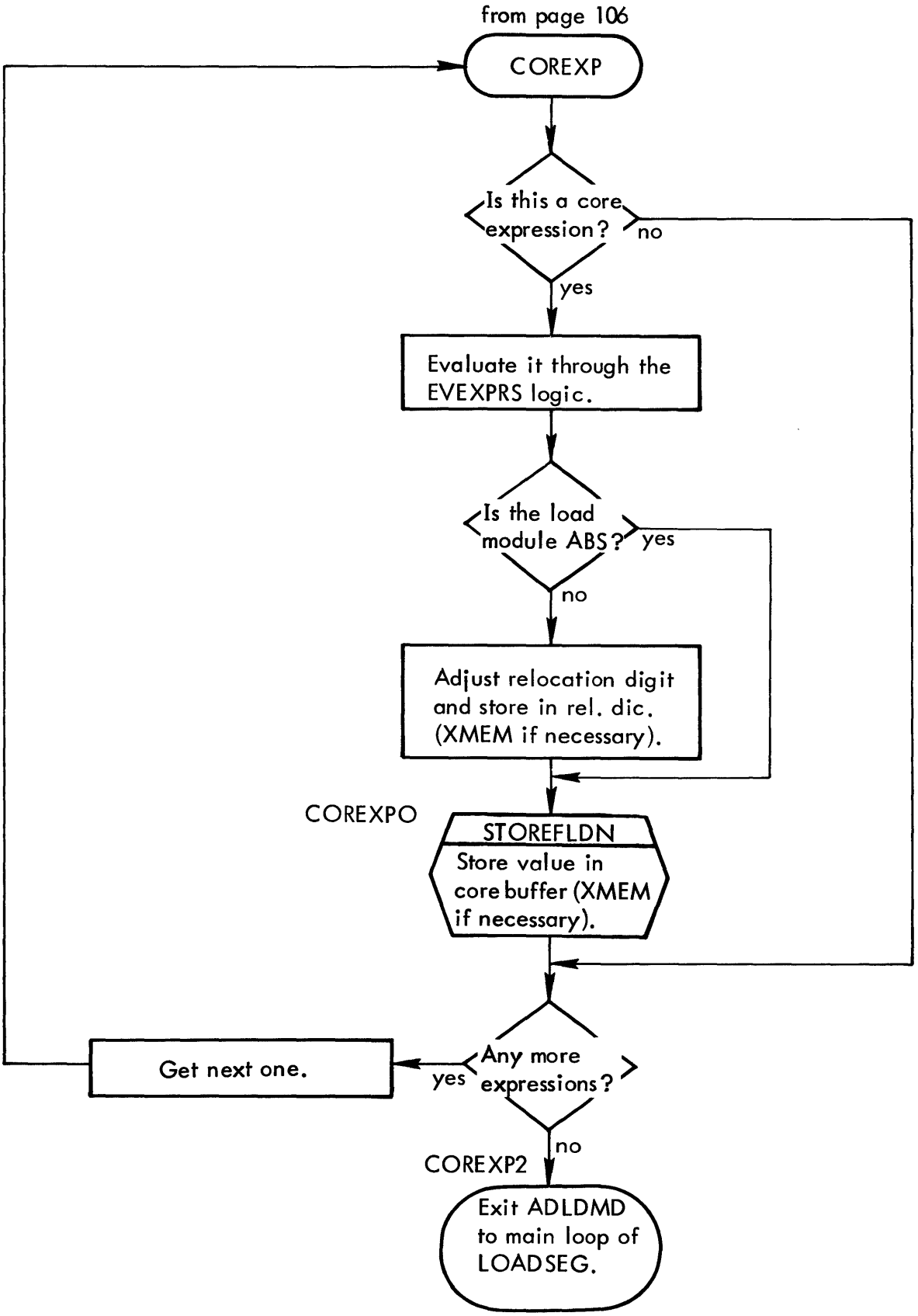


Figure 28. PASS2 Load Module Processor Flow Chart (cont.)

7.0 WRITING THE LOAD MODULE (WRT)

WRT is entered at WRITESEG from PS2. If this is a library load module, it updates the library dictionary (WRITELIB), makes a copy of the REF/DEF stack for mapping purposes (LIBCPY), and eliminates forward references from the just copied stack (SQZ). The segment's stacks are written (WSEGL) as are the segment's core images and relocation dictionaries (WSEG1).

WRT performs several additional tasks after the root has been constructed. If extended memory mode is in effect and a standard load module is to be constructed, the pages of all of the segments are put together and written out (XMEM). If a paged load module is to be constructed, the first record of each overlay segment's 00, 01, and 10 areas are shortened and the first few records of the root are read into core to insert the appropriate tables (SUPMEM). In any case, once the root has been processed, the DCB Name Table is built (SAVEROOT) as well as the DCBs and TCB (FIXROOT). The HEAD and TREE records are constructed and written to the load module file.

See Figure 29 for an overall view of the flow of WRT.

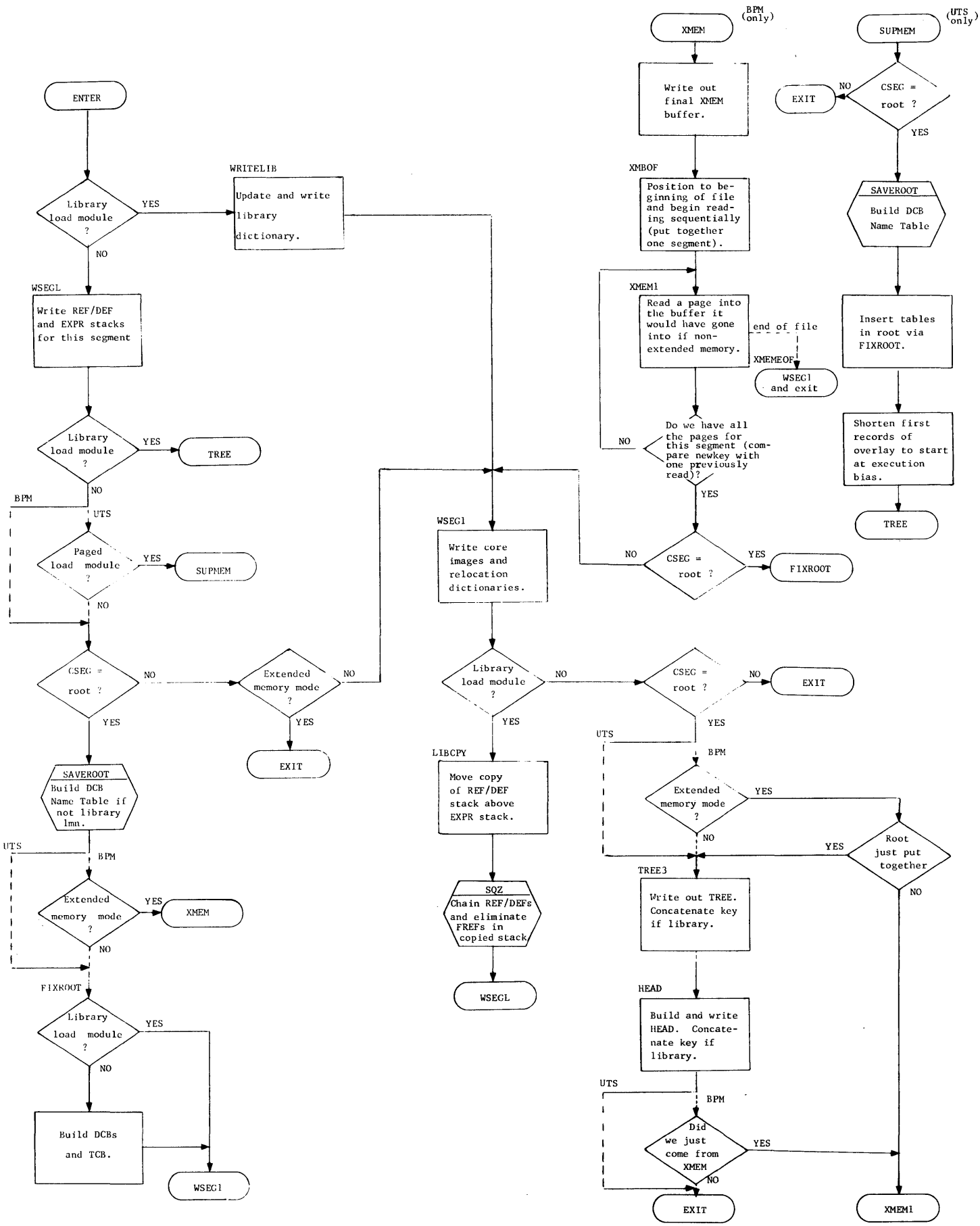
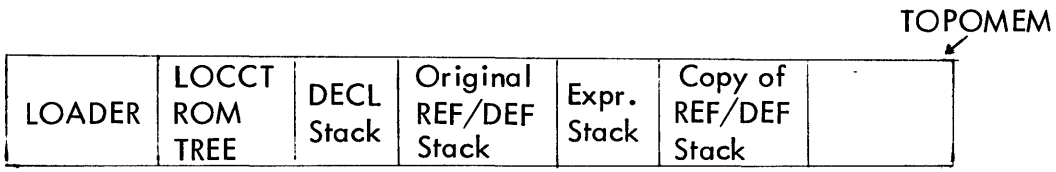


Figure 29. WRITESEG – Overall Flow

LIBCPY: The MAP routine in FIN must have an unchained, expendable copy of the library load module's REF/DEF stack. For this reason LIBCPY makes a copy of the REF/DEF stack above the expression stack before the SQZ routine is entered. In order to provide as much room as possible for this copy, the library load module's core image and relocation dictionary records are written out immediately prior to entering LIBCPY.

After LIBCPY has moved the REF/DEF stack above the expression stack, memory layout is as follows:



A TREE pointer is adjusted so that the new copy of the REF/DEF stack is squeezed, chained, and written out by WRITESEG, and the original REF/DEF stack is used by the MAP routine (which needs the area above the REF/DEF stack for sorting names). The start of the original REF/DEF stack is remembered in MBIAS.

Entry is made to LIBCPY from the WSEG1 routine. After calling SQZ, LIBCPY exits to WSEGL, whereupon the proper stacks are written out.

SQZ: This routine streamlines a library load module's REF/DEF stack in order to expedite subsequent adding of a library to a user's load module. Two functions are performed:

a. at RFDFLOOP – all evaluated forward reference entries in the REF/DEF stack with bit 10, word 0 reset are removed from the stack. All

evaluated expressions are removed which involve that entry. If bit 10 is set, the FREF entry is retained (so that an unevaluated DFREFH expression in the library load module involving this FREF (add FREF) can be evaluated when it is merged into another program).

b. At SQZDN – chaining is installed. The VALUE word of every REF/DEF entry becomes the head of a chain within the expression stack which replaces pointers to the REF/DEF entry; the tail of the chain = 0. That is, the VALUE word of each REF/DEF entry is replaced by a pointer to the word in the expression stack that formerly pointed to that REF/DEF entry. (The pointer is a displacement relative to the base of the expression stack.) This expression stack word is replaced by a pointer to the next user of the REF/DEF entry. This process continues until a zero terminates the chain.

For example, consider a DEF entry which has a displacement of X'B'

words into the root's REF/DEF stack. Then any expression involving the DEF refers to it by means of a pointer of the form X'B'. Assume there are three such pointers in the root's expression stack: PT1, PT2, and PT3, with displacements X'F', X'1A', and X'22', respectively, relative to the base of the expression stack. Then the chaining process with respect to this DEF entry is outlined as follows:

Word	Displacement	Contents Before Chaining	Contents After Chaining
Value of DEF entry	X'B' (Into R/D stk)	Constant or Addr	X'F'
PT 1	X'F' (into expr. stk)	X'B'	X'1A'
PT 2	X'1A' (into expr. stk)	X'B'	X'22'
PT 3	X'22' (into expr. stk)	X'B'	0

The benefit is that the expressions do not have to be relocated (with respect to the new REF/DEF stack) each time the library load module is added to another.

WRITELIB: Writes the dictionary for the library. This entails three cases. The three cases are distinguished via abnormal or nomal returns. In any case, a ROM of the same name as the LMN is deleted to insure proper handling.

Case 1 - The library (:LIB and :DIC) do not exist. Here we create them by opening in the OUT mode.

Case 2- The library exists but this new load module is not within it.

Case 3 - The library exists and this new load module is to replace one with the same name which already exists within it.

In general, WRITELIB does the following:

Step 1. Opens the :DIC file; and then it opens the library file with the load module name synonymous to :LIB. The only anticipated abnormal return would be that the file :LIB does not exist (Case 1) and we go to FIRSTLIB. The file :LIB is created and the opening is reattempted with the load module name, and we proceed to Step 3.

Step 2. (RDRFDF) If :LIB already existed, an attempt is made to read the REF/DEF stack from :LIB for a module with the same name as our module. An error return implies that the desired load module is not within :LIB and we proceed to Step 3. If the read is successful, a delete CAL is made to the :DIC file, with each DEF serving as a key to remove the old module's dictionary entries. (A delete CAL is also made for each DDEF and DSECT entry in the old REF/DEF stack).

Step 3. (WRITEDEF) Then, we run through our module's REF/DEF stack. Every DEF, DDEF, and DSECT of our module is used as a key to write the :DIC file, the record being the module name. The dictionary is closed and we exit to WSEGL of WRITESEG.

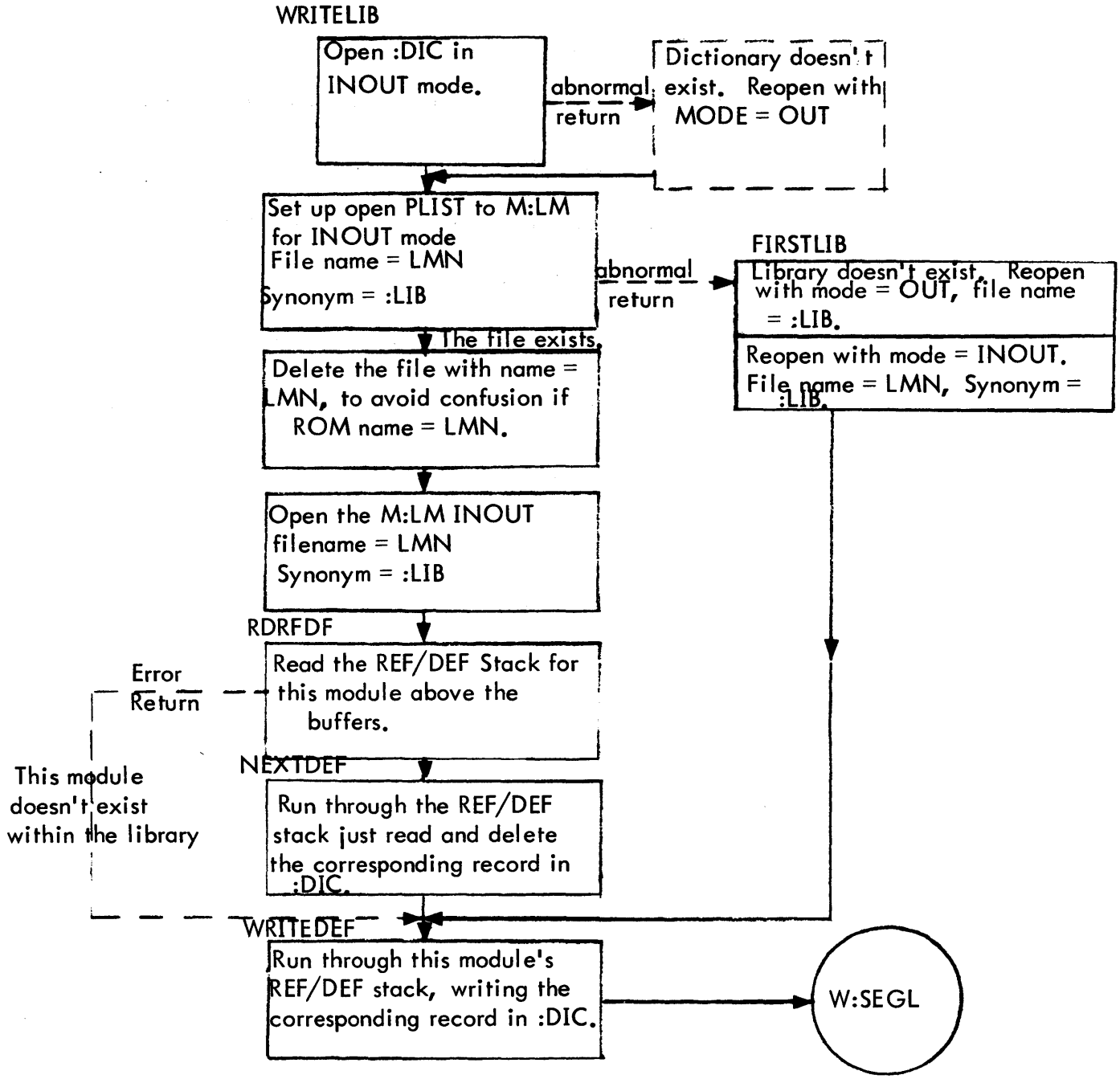
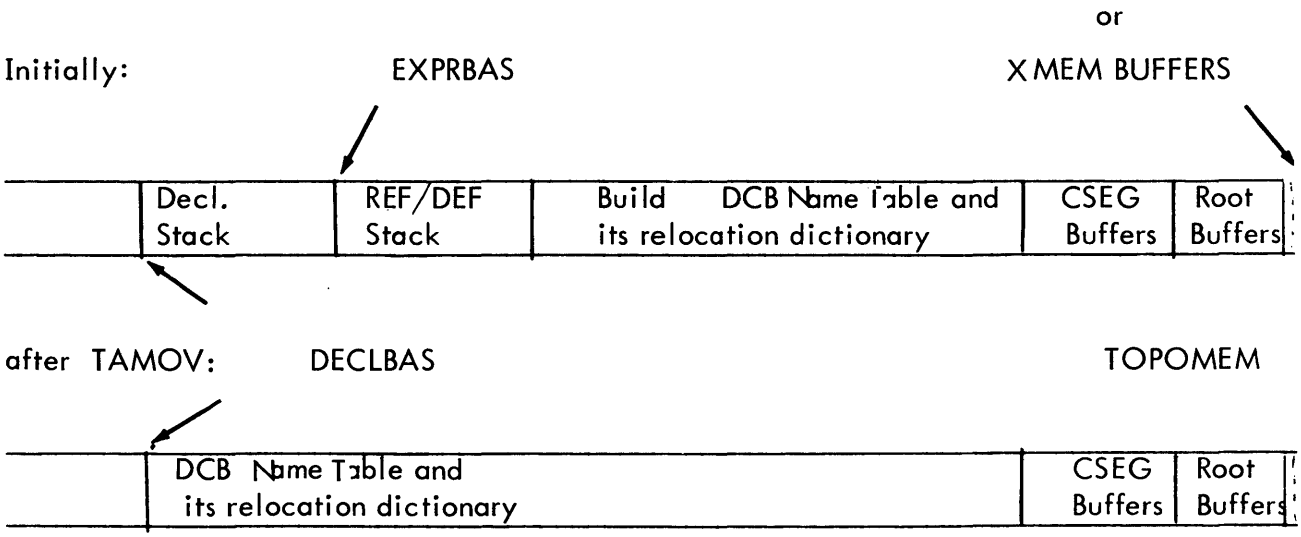


Figure 30. WRITELIB Flow Chart

SAVEROOT constructs the DCB Name Table and its relocation dictionary. Entries are put in this table for every DSECT with a name beginning with M: or F: in the REF/DEF stack. The DCB Name Table is initially built above the REF/DEF stack (beginning at the expression stack). Because the DCB Name Table was the only item requiring the REF/DEF stack after mapping the stack can now be destroyed and we move the DCB Name Table down to the declaration stack (at TAMOV). This is done to make as much room as possible for the reconstruction of XMEM files, if necessary. (If there is no room to build the table, i.e., we would collide with the buffers, loading is aborted).



Recall that at the end of PASS1, all PREF DCBs were set to type B. During the building of the DCB Name Table, SAVE-

ROOT flags the location word (bit 8) of each table entry which resulted from a type B REF/DEF entry, as a signal for DCB building in FIXROOT .

FIXROOT: Moves the TREE into the 01 buffer. Then it moves the DCB Name Table and its relocation digits to the buffers (01 for BPM, 10 for UTS). The TCB and its relocation dictionary are built in the 00 buffers. The DCB Name Table is scanned for those DCBs which are to be built by the Loader. If one is to be built (we know this from the hi-order flag bit in the location word of the entry), FIXROOT builts it (in 01 for BPM or 10 for UTS) then checks whether it has a standard name. If so, default information is inserted. The proper relocation dictionary is built. In BPM, if the load module is being written to private disk pack, the serial number(s) is inserted in the M:SEGLD DCB.

XMEM: (BPM only) This routine is entered only if CSEG = root segment and extended memory mode is in effect, and a standard load module is being constructed. Its function is to reconstruct the load module from the XMEM file into the form necessary for writing it out as a keyed file in load module format. This requires that the pages be placed in the core image and relocation buffers, (see Figure 9b). Re-

call that the keys of idX indicate the page number of the buffers (see Figure 24).

The last core buffer is forced out and, if the module is not ABS, the last relocation dictionary buffer is forced out. The file is positioned to its beginning and is read sequentially, first with 0 byte count to get the next key from which the buffer address is calculated and again to read the page in. This process continues until an end-of-file is encountered or the segment number in the key changes. If the segment read is the root segment, FIXROOT is called. The segment is then written out into the normal load module file. This process continues until all segments are reconstructed. Notice that the advantage afforded to large load modules by XMEM during this concatenation process is that the area of core otherwise dedicated to the stack can now be part of the 6 buffers.

A final constraint on the size of the load module that can be concatenated is that the DCB Name Table and its relocation dictionary (which have been temporarily placed at DECLBAS and up by SAVEROOT) are co-resident with the largest segment.

SUPMEM: If extended memory mode is in effect and a paged load module is being constructed, SUPMEM is entered immediately, after the current segment's stacks are written. If the segment is not the root, return is made to ENDWRT1 whereupon WRITESEG exits. If CSEG is the root, the following functions are performed.

The last core image record (from EVL) is written out and SAVEROOT is called to build the DCB Name Table. Upon returning from SAVEROOT, the size of the root's tables are determined and, in the order of protection types, GETRECS is called to read in the records which are to contain these tables (if the records already exist).

GETRECS reads in the first such record into the first available page above the DCB Name Table. The next record is read just above this page, and so on. If GETRECS tries to read a record which does not exist, it still reserves space for this record in the next available page. Finally, the buffer pointer corresponding to the protection type of the records being read in (RSEG00, RSEG01, or RSEG10) is adjusted to point to the beginning of these records as they sit in core.

After these records have been collected, FIXROOT is called to build and insert the table. The updated records are then written out.

Next the first record of the 00, 01, and 10 areas of each overlay segment if it does not begin on a page boundary (and the root's 00 area, if it is a core library, has been associated) must be shortened to start at the first word of code. This is done by reading each record into the page at the top of memory (as a 512-word record). The size of the record to be output is computed from

the execution bias in the segment's TREE. The buffer pointer is moved accordingly and the truncated record is written out.

When this process is complete, SUPMEM exits and the HEAD and TREE records are constructed.

8.0 FINISHING UP (FIN)

The FIN segment comprises the final stage of the Loader. By now the entire load module has been written out. All that remains is to output the severity level, perform any modifications per !MODIFY cards, and generate the load map.

FIN is entered from LDR at FINISH. FINISH computes and outputs the severity level. At this point the user sees the general allocation summary and the severity level. Next the MOD routine is called. This routine establishes the environment for both the MODIFY (Catalog Number 705396) and MAPER routines. If the severity level is less than or equal to the maximum (supplied by the user or CCI in the LOCCT), modifications are performed per the !MODIFY cards which have been packaged into the idD file by CCI. In any case MOD calls MAPER to generate the load map.

MOD first checks to see if: 1) a library load module is being formed; 2) extended memory mode is in effect; or 3) the severity level is greater than the allowable maximum. If any of these conditions are true, a flag (N01DD) is set to inhibit modifications. Otherwise the idD file is opened (if the file doesn't exist, N01DD is set). The REF/DEF stack for the first segment is read (except for a library load module, whose stack is already in core).

Now if N01DD $\neq 0$, this segment is mapped and the next segment's REF/DEF stack is read. If N01DD = 0, the core images and relocation dictionaries for this segment are read, the idD file is read, the MODIFY routine is called to perform the modifications, the segment is rewritten, and the load map is generated by MAPER. This processing continues until there are no segments, whereupon MOD exits to the main FINISH program.

At this point, if the severity level is greater than the maximum allowable, the loader aborts. Otherwise FIN closes M:LM and M:LL (load module and map DCBs) with SAVE and returns normally to the driver in LDR (which exits to CCI or PASS3).

See Figure 31 for a flow of the FIN segment.

MAPER: The MAPER routine works mostly within the framework of the REF/DEF stack itself in order to generate a segment's load map. The routine does, however, use the core above the REF/DEF stack for two purposes: 1) to save "displaced" DEF entries (a DEF whose defining expression is located in another segment) so they can be included in the map of the segment in which they are defined; 2) to collect sort keys (a pointer to a REF/DEF entry) of all the names of a particular type (e.g., SREF, DEF, PREF) in order to produce an alphanumerically sorted name list. The displaced DEF stack is saved throughout the entire load module mapping process and is constructed from TOPOMEM down. The sort keys are destroyed after each name list is written (the keys are built just above the REF/DEF stack). Any possible collision between these areas results in halting the addition of more sort keys/displaced DEF entries. See Figure 32 for memory layout during MAPER.

MAPER first outputs several lines of preliminary information which it obtains from this segment's TREE (and the LOCCT table, if this is the root segment). The boundaries and sizes of the protection type areas are computed by the SEGEVAL routine and translated and output by VALMOVE. Then four major routines – PREPROC, PSMALIST, SORTMAP, and MAPLIST – are called in succession to generate the name lists.

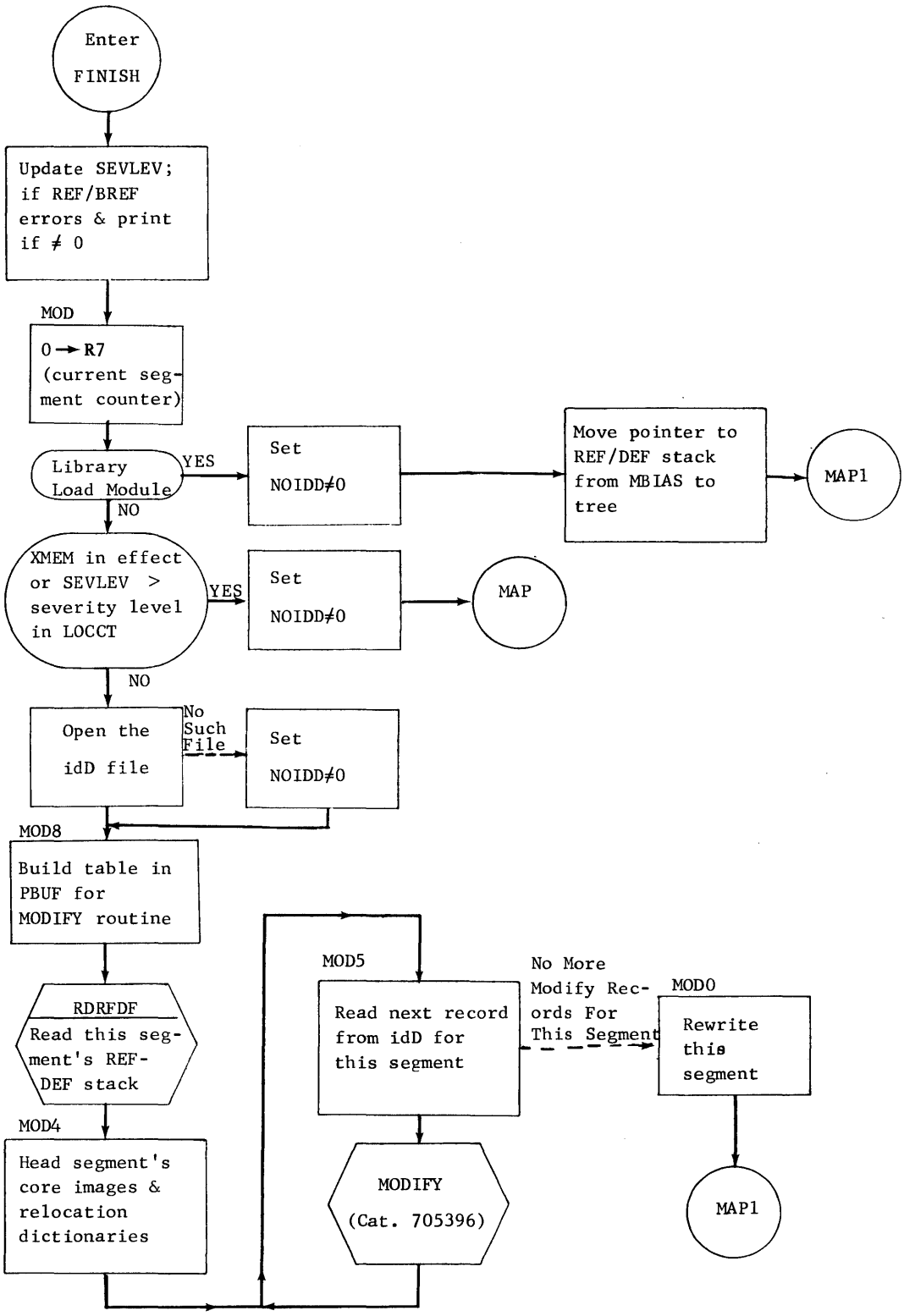


Figure 31. FINISH Flow Chart

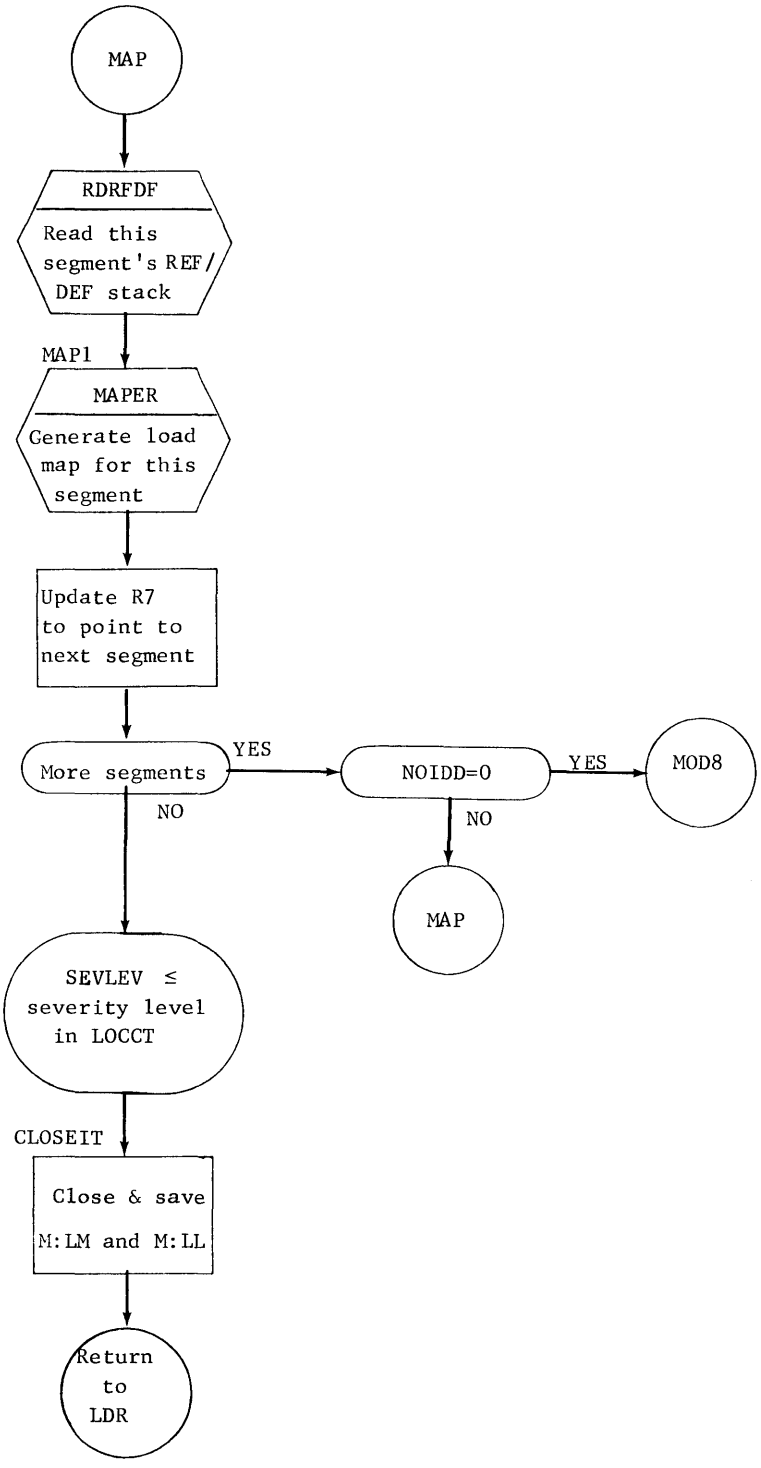


Figure 31. FINISH Flow Chart (cont.)

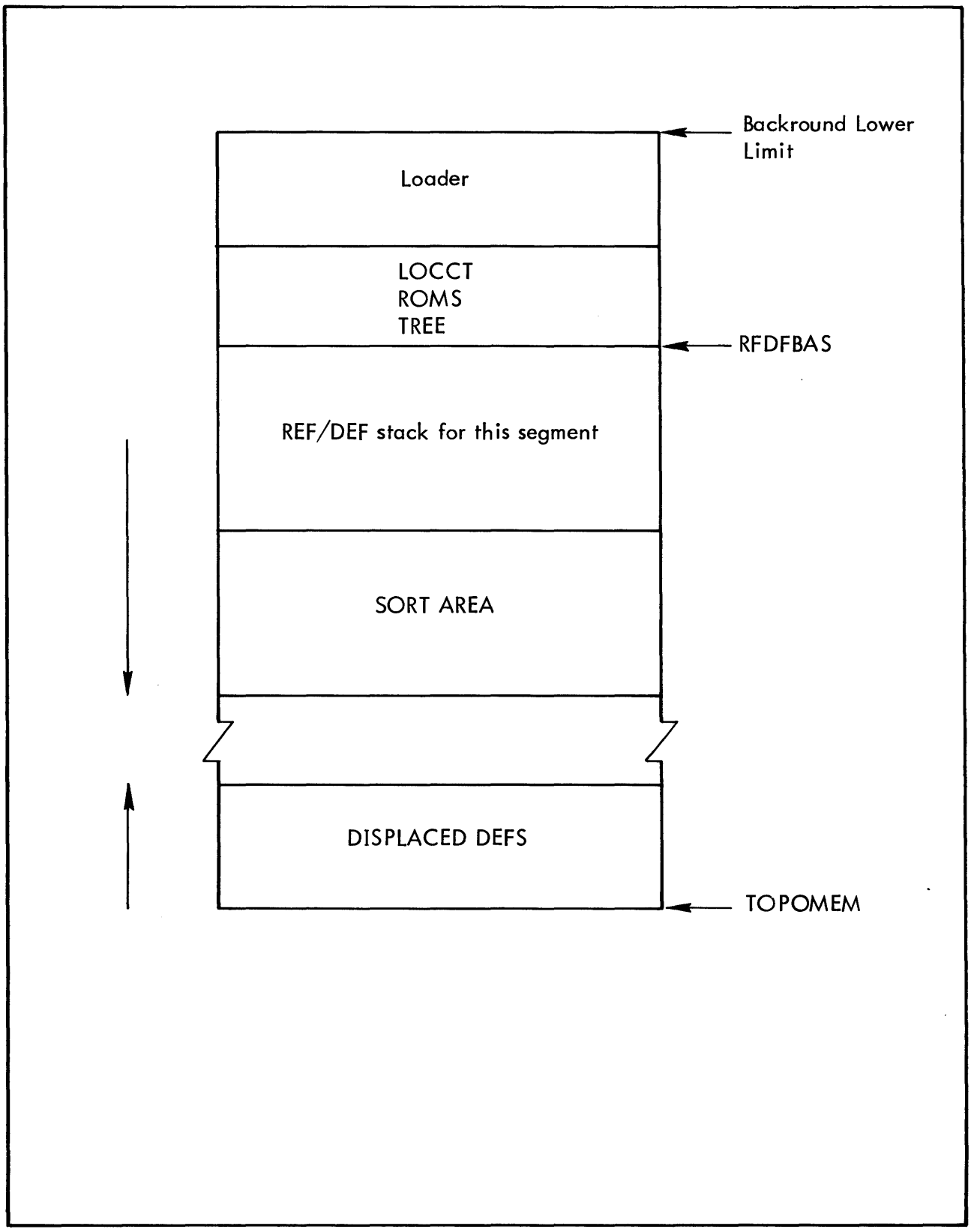


Figure 32. Memory Layout During MAPER Routine

PREPROC runs through the REF/DEF stack, deleting unnecessary REF/DEF entries (FREFs and control sections with zero size), clearing the resolution word of each entry (used for chaining the stack in SORTMAP), flagging ADEFs, and resolving relocatable values to word resolution with a byte displacement (of the form X'0B0AAAAA'). In addition, each displaced DEF entry in this stack is moved to the displaced DEF stack and deleted from this stack. After the entire REF/DEF stack has been scanned, the displaced DEF stack is examined for any entries belonging to this segment. If any are found, they are appended to this segment's REF/DEF stack.

MAPER calls PSMALIST four times – each time to generate a list of a specific type of REF/DEF entry. In this way PSMALIST produces the PREF, SREF, DDEF, and ADEF lists (no list is generated if the stack is void of that type of entry). PSMALIST scans the REF/DEF stack for a given type of entry, building sort keys for all the entries of this type it finds. Then SSSUBR is called (via SRTEXIT2) to perform the sort and MAPFIN3 is called to list the names.

SORTMAP uses the resolution words to chain (in order of ascending value) either: 1) all CSECTS, DSECTS, and relocatable DEFs if (MAP, VALUE) was specified on the LOAD card, or 2) all CSECTs, the first relocatable DEF in each CSECT, and all DSECTs if (MAP, NAME) was specified. Also, if NAME was specified, SORTMAP builds the sort keys for the relocatable DEFs in the sort area and calls SSSUBR to sort them.

MAPLIST directs the generation of the relocatable DEF list. After the correct heading is printed, the chain through the REF/DEF stack is followed to move each

entry to the output buffer. Whenever MAPLIST encounters a control section as the next link, it calls NUSECT to write out the current line and move the control section's information to the buffer. If the NAME option was specified, MAPFIN3 must be called to sort and output the DEF names.

Note: The sort routine implemented is that described in a "A High-Speed Sorting Procedure", D.L. Shell, Communications of the ACM, Vol. II, July, 1959.

APPENDIX A

LOADER-GENERATED INTERNAL SYMBOL TABLES (UTS ONLY)

PURPOSE: To output internal symbol table (IST) records as a part of a load module.

DEFINITIONS: A source program can contain both internal and external symbols. An external (or global) symbol is one which is declared as a DEF in this program and which may be referenced in other, separately assembled programs as a REF or SREF. An internal symbol is one which applies only within the given source program (and hence is not REF'd or DEF'd). A symbol table consists of a list of correspondences between symbols used in a source program and the values or virtual core addresses assigned to them by the Overlay Loader (or LINK).

USAGE: The association of internal and external symbol tables with a user's program enables the user to reference such symbols under various debugging processors (in particular, under DELTA). Under DELTA, the user can operate on his programs in what appears to be assembly language symbolic; with regard to internal symbol tables, he has the ability to define which set of internal symbols are to be used for specific debugging activities.

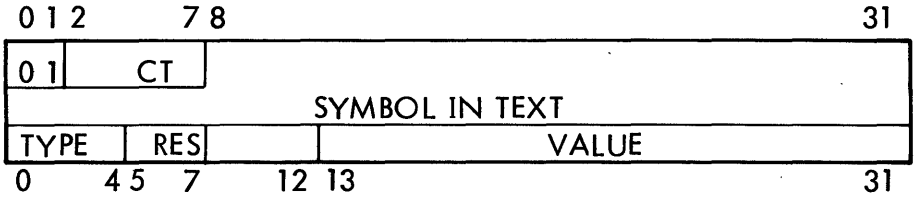
COMMENTS: The Loader builds internal symbol tables only. (Global symbol tables can be generated by the SYMCOM processor.) Each IST corresponds to one particular ROM. If more than one ROM is contained in an element file, an IST is generated for only the last ROM in the file. IST generation is suppressed for library load modules and core libraries (i.e., load modules whose name begins with :P).

INPUT: The Loader generates an internal symbol table entry when it encounters a "Type and EBCDIC for Internal Symbol" load item (control byte X'12') in a ROM. See BPM Reference Manual, Appendix A, for the format of this load item.

OUTPUT:

The loader outputs one IST record for each element file (specified in the EF list) which contains a ROM with IST load items. The record is a keyed record, the key consisting of the element file name concatenated with X'10'. The internal symbol table has two types of entries – symbols whose values are constants and symbols whose values are addresses.

SYMBOL TABLE FORMAT-ADDRESS TYPE



where

CT = Character count of the original symbol.

SYMBOL = The first 7 characters of the symbol. Symbols with fewer than 7 characters are zero-filled. Longer symbols are truncated to 7 characters, though the original character count is retained.

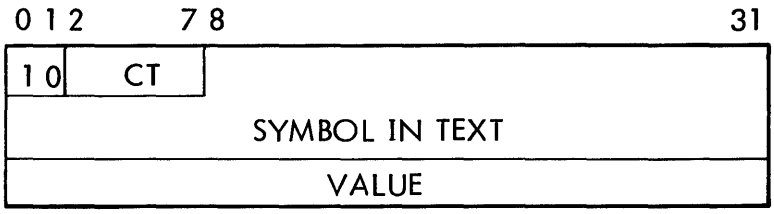
TYPE	00000	Instruction
	00001	Integer
	00010	Short floating point
	00011	Long floating point
	00110	Hexadecimal (or packed decimal)
	00111	EBCDIC text (or unpacked decimal)
	01000	Logical array
	01001	Integer array
	01010	Short floating point array
	01011	Long floating complex array
	10000	Undefined symbol

RES is a 3-bit field indicating the internal resolution.

000	Byte
001	Halfword
010	Word
011	Doubleword

VALUE is the address corresponding to this symbol, in byte resolution.

SYMBOL TABLE FORMAT – CONSTANT TYPE



where

CT and SYMBOL are the same as above.

VALUE is the 32-bit constant value.

FLOW: INIT1 checks to see if the (PERM, LIB) option is specified or if the LMN name starts with :P. Either of these conditions results in setting SYMBOLTB to -1.

Initialization of the IST buffer (to be used for IST generation during PASS2) occurs in INIT2. Space is allotted for the buffer between the expression stack and the core image/relocation buffers. Refer to Figures 8 and 9a for the Loader's memory layout during PASS2. (The table is constructed from the top end of the buffer down.)

Internal symbol tables are constructed in the LP1 section of LOADSEG. When a X'12' control byte is encountered at LDR1, a branch is made to SD12. If SYMBOLTB is non-negative, LPT checks the IST buffer limits to determine if the expression stack has grown into the IST being constructed (by the addition of core expressions) or if the addition of a new IST entry would cause a collision with the expression stack. If either of these events occur, IST generation is suppressed (SYMBOLTB is set to -1). Otherwise, the new entry is constructed and added to the IST, and SYMBOLTB is updated to point to the base of the new entry.

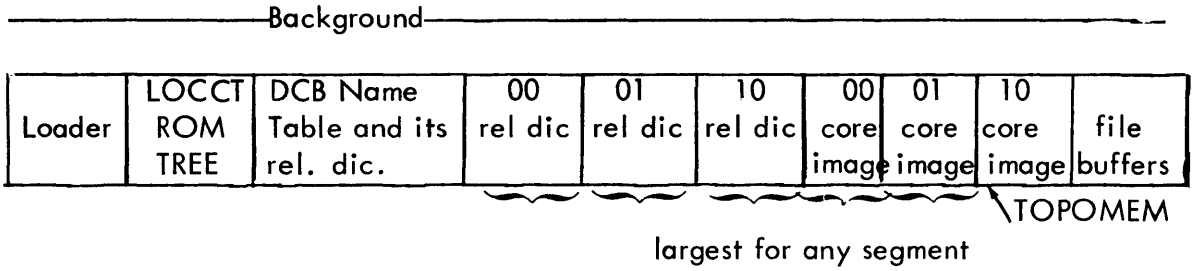
At module end the current IST is written out. If this is the largest IST output so far, its base address (in SYMBOLTB) is remembered in BSEG2. In any case SYMBOLTB is reinitialized to the first word above the IST buffer (kept in SYMTOP).

In WRITESEG, the size of the largest IST is stored in Word 8 of the HEAD record (as well as its future location under DELTA).

SUMMARY OF LOADER RESTRICTIONS

1. Load Module Size

The primary constraint with regard to the largest standard load module that can be constructed by the Loader concerns the number (Background size - Loader size file buffers - LOCCT, ROM and TREE Tables.) This represents the maximum size of that area which must contain the DCB Name Table and its relocation dictionary plus the largest core image of each protection type (00, 01, and 10) of any segment and their respective relocation dictionaries.



An additional constraint for a standard load module – and the main constraint for a paged load module – is that there must be enough room (in Pass Two) to accommodate the Loader, the LOCCT, ROM, and Tree Tables, the maximum declaration, REF/DEF and expression stacks, plus 2 pages for building the load module (or 1 page if the module is to be ABS.)

2. The name of an input file must be $\leq$ 10 characters (see ROM Tables).

3. The name of a load module must be $\leq$ 11 characters (see LOCCT Table).

4. If a DEF in a library load module is > 11 characters, the corresponding entry in the :DIC file is forced to 11 characters. (The DEF entry in the library load module itself is not changed.)

5. A load module acceptable for the combination with ROMs to form a new load module must be of one protection type, relocatable, and not overlaid. DSECTs in such a load module are allowed only if the entire load module consists of one DSECT.

6. A load module will be set ABS if any of the following conditions exist:

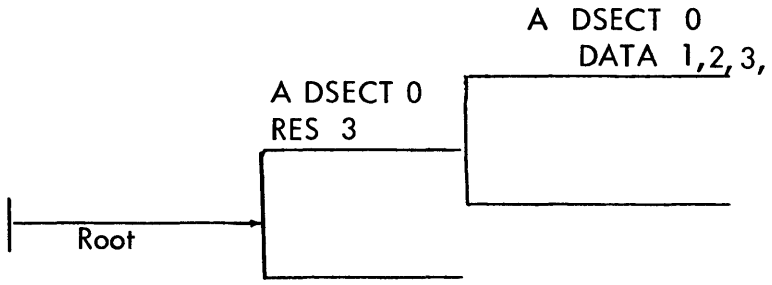
   a) It contains a relocatable field not ending on a half word boundary.

   b) It contains an expression of mixed resolution.

   c) REF or BREF has been specified on the !LOAD card.

7. Segments may communicate with each other via REFs and DEFs only if they lie in the same path.

8. Load items of a DSECT are always placed in the corresponding DSECT of the root segment. That is to say, there must be a DSECT by the same name in the root. The following case is not permitted.



9. The loader cannot perform modifications (!MODIFY) on a library load module. That is, a !MODIFY following a !LOAD (PERM, LIB) will be ignored.

10. The loader will ignore modifications (!MODIFY) if extended memory mode has been entered.

11. If a low segment references a DEF name which is both in a higher segment and a library, the library DEF will be used.

COMMON QUESTIONS ABOUT THE LOADER

1. Why is the expression stack retained as a permanent part of the load module?

   The expression stack is retained for only one reason: that is, for the purpose of combining the load module with other ROMs. At the time of combination, we must process the unevaluated core expressions to complete the load items which involve PREFs. The PREFs will presumably have been satisfied and the expressions involving them will not be evaluatable.

2. What are the final contents of the expression stack?

   The final contents consist of:

   a. Defining expressions for DEFs and forward references. (If this is a library load module, only those expressions involving unsatisfied forwards are retained. The others are squeezed out as are the REF/DEF entries which identified the forward numbers.)

   b. All unevaluated core expressions (core expressions are unevaluatable if they involve PREFs).

3. Load modules which are combinable with ROMs can have only one protection type. Why is this so?

   Generally speaking, load modules are relocated by computing a relocation factor (=new bias-module bias). This relocation factor is added to all relocatable items in the module. (The relocation factor is actually modified via the relocation digit to the proper resolution but this is irrelevant for the current discussion.)

   Consider a load module with two protection types.

   If we try to combine this load module with other ROMs we must also relocate the

core images (00 and 01) with respect to their newly acquired position in the target load module. Having detached the 00 and 01 areas we have of course changed the relative distance from one to another and now cannot compute a relocation factor since "module bias" is meaningless.

Example: Consider a load module, X, with two protection types 00 and 01. The instructions at α are in 01 and ZAP is in 00.

```
α  LW, 1      ZAP
   LI, 1      $
```

Assume that in X, 00 begins at relative location 0 and 01 begins at relative location 500. Assume that ZAP is relative location 100 and α is relative 550.

Now assume that for the new module, X', the new positions for 00 and 01 are to begin at 2000 and 4000, respectively.

The Loader sees only the core image from X:

```
(4050)  LW, 1   100
(4051)  LI,  1  551
```

It has no way knowing that it should relocate for X' by adding 2000 to α but 3500 to $\alpha + 1$.

APPENDIX B

STORAGE LAYOUT OF STUFF

NAME	DISPLACEMENT	CONTENTS
DECLSTK	+0	Declaration stack pointer doubleword.
DECLSTK1	+1	Declaration stack pointer doubleword.
RFDFSTK	+2	REF/DEF stack pointer doubleword.
RFDFSTK1	+3	REF/DEF stack pointer doubleword.
EXPRSTK	+4	Expression stack pointer doubleword.
EXPRSTK1	+5	Expression stack pointer doubleword.
DECLBAS	+6	Base of declaration stack; see Figure 9b.
RFDFBAS	+7	Base of REF/DEF stack.
EXPRBAS	+8	Base of expression stack.
BSEG1	+9	Temporary segment number; used for small subroutines as in INIT1.
BSEG2	+10	Base address of largest internal symbol table.
CSEG1	+11	Displacement from beginning of tree tables to beginning of tree for current segment.
CSEG2	+12	Temporary storage for renumbering current segment number; used in PS1 for temporary sequence number in the name routines.
CROM1	+13	Current ROM pointer in ROM table; displacement from start of ROM table to current ROM; used in PS1 and EVL.
CROM2	+14	Temporary storage for current ROM pointer; used in PS1.
CRFDF1	+15	Pointer to the current REF/DEF entry being looked at.
CRFDF2	+16	Top of REF/DEF stack being looked at.
CURBYTE	+17	Displacement into card image now being read in GBYTE; contains last byte read in ROM record.
RECDSIZE	+18	Size of ROM record just read by GBYTE.

NAME	DISPLACEMENT	CONTENTS
SEQNUM	+19	Actual sequence number of record just read in GBYTE routine.
SEVLEV	+20	Severity level of load module; starts out with that of ROMs; gets raised if need be; see WRT and FIN.
XSL	+21	Maximum severity level from !LOAD card; now in LOCCT.
LASTCARD	+22	Flag that this is last card of this ROM; see GBYTE.
BUF	+23	Used in PS1 and EVL as input buffer for reading ROMs; used as output buffer by FIN; used as buffer for the map; some of its words are used by WRT.
BUF2	+53	Used to construct an expression from load relocatable type load item; see pages 98 and 99; used in WRT.
TEMPPTR	+57	Used to keep track of temp stack in user's TCB; see INIT2, ALL, WRT.
TREEPTR	+58	Pointer (execution type address) to loader – built tree table; used in WRT.
ERRPTR	+59	Pointer (execution type address) to loader – built ERTABLE; used in WRT.
FCOUNT	+60	Size of DCB name table; used in PS1.
FTABLE	+61	Starting address of DCB name table at execution time.
ERRTAB	+62	ERTABLE size from the LOCCT.
ERRSTK	+63	ERSTACK size from the LOCCT.
TCBSIZE	+64	Total size of target TCB including ERSTACK and ERTABLE sizes; see WRT and IN2.
TCBPTR	+65	Execution starting address of target TCB; see ALL and WRT.
FTAB	+66	Starting address of DCB name table at execution time; set in ALL; used in WRT.
RSEG00	+67	Pointer to root segment for protection type 00; see Figure 8.
RSEG01	+68	Pointer to root segment for protection type 01; see Figure 8.
RSEG10	+69	Pointer to root segment for protection type 10; see Figure 8.

NAME	DISPLACEMENT	CONTENTS
RREL00	+70	Pointer to root segment's relocation dictionary for protection type 00; see Figure 8.
RREL01	+71	Pointer to root segment's relocation dictionary for protection type 01; see Figure 8.
RREL10	+72	Pointer to root segment's relocation dictionary for protection type 10; see Figure 8.
CSEG00	+73	Pointer to current segment for protection type 00; see Figure 8.
CSEG01	+74	Pointer to current segment for protection type 01; see Figure 8.
CSEG10	+75	Pointer to current segment for protection type 10; see Figure 8.
CREL00	+76	Pointer to current segment's relocation dictionary for protection type 00; see Figure 8.
CREL01	+77	Pointer to current segment's relocation dictionary for protection type 01; see Figure 8.
CREL10	+78	Pointer to current segment's relocation dictionary for protection type 10; see Figure 8.
MAX00	+79	Largest protection type areas which have to be allocated for each segment; see INIT2, FINDLGSTPATH.
MAX01	+80	Largest protection type areas which have to be allocated for each segment; see INIT2, FINDLGSTPATH.
MAX10	+81	Largest protection type areas which have to be allocated for each segment; see INIT2, FINDLGSTPATH.
DLOC	+82	Execution location counter for 00.
PLOC	+83	Execution location counter for 01.
SLOC	+84	Execution location counter for 10.
LOC	+85	Load location counter; see EVL.
START	+86	Starting address; gets put in HEAD; see DSTART in EVL.
LOCCT	+87	Address of LOCCT, first available page above loader's procedure.
LOADBAS	+88	Actual load bias; either from LOCCT or defaults to BGLL; see INIT2.

NAME	DISPLACEMENT	CONTENTS
MODBAS	+89	Used for merging core image record into XMEM buffers; see EVL.
RELDBAS	+90	Base of relocation dictionary for core image library; used in EVL.
MBIAS	+91	In WRT, start of original REF/DEF stack.
FBIAS	+92	Used for paged load modules; address pointing into loader's core image buffers; see ORG in EVL; see also WRT's old XMEM code.
BIAS	+93	Equivalent of ORG to execution address of start of ROM.
RDIG	+94	Relocation digit; see ADLDMD.
MODSIZ	+95	ARS from M:EF after reading relocation dictionary; see ADLDMD in EVL.
NOTLLM	+96	Flag in WRT for not a library load module.
MAXRFDF	+97	Computed dynamically in PS1 to find longest REF/DEF path needed by PS2.
MAXEXPR	+98	Computed dynamically in PS1 to find longest expression path needed by PS2.
TOPOMEM	+99	Last available address (ends in E).
OPENEF	+100	Contains the open PLIST for M:EF.
OPENDIC	+117	Contains the open PLIST for :DIC.
PBUF	+132	Print buffer for loader diagnostics.
CSECFLG	+153	Flag for special CSECT used in merging library lmns; see ADLDMD in PS1.
PLIB	+154	Flag which gets set if addition of a core expression would cause expression stack to overwrite a core image buffer above it; see EXPRIN routine in EVL.
LIB	+155	1 if a library lmn is being added; see ADLDMD.
XMKEY	+156	Extended memory mode key used to write core image records; initialized in INIT2; used in EVL and WRT.
LOCWD	+157	First word of the LOCCT, containing parameter bits.
USID	+158	User ID number passed in register by Monitor; used in IN1 and IN2 to open temporary file.

NAME	DISPLACEMENT	CONTENTS
RFLDSG	+159	See REF/BREF option; segment number of where DEF is defined in the branch referencing mode.
ERFLAG	+160	Message key; see MESSAGE.
MXRFDFSG	+161	Contains segment number; aid in determining path having largest REF/DEF stack; see PS1.
NXTAVPG	+162	Execution address of page just above the load module; gets put in HEAD; computed in IN1; picked up in WRT.
RLOC	+163	Loader's load location counter for relocation dictionaries; goes with LOC.
O1SIZ	+164	For special CSECT in merging library lmns; see ALL.
TRESIZ	+165	Size of the loader to see if it must do SEGLOADs or can just branch; see LDR.
FCOMSIZ	+166	Size of blank COMMON from the LOCCT; set up in PS1 when loader finds ROM defining F4COM; takes largest size for any DSECT declaration with name or F4COM.
XMRKEY	+167	Extended memory mode key for reading the relocation dictionary.
04LOC	+168	In ALL pointer to remember last control section when searching for special library control sections; in SPECDSEC location of an 04 entry.
DOREFPTR	+169	In BREF mode pointer to name S:OVRLY in REF/DEF stack; see IN1.
RFLDTBSZ	+170	REF count from LOCCT, word 4.
BREFERR	+171	Count of REF's overflowing table; if in BREF mode, count of nonbranching REF's overflowing table; BREF error in EVL is picked up in FIN.
PASS3RET	+172	Information saved for PASS3/CCI if must return to it; see INIT1 and LDR.
ENTFLAG	+173	Type of entry we are making: PASS3 or CCI; see INIT1 and LDR.
CORELIB	+174	UTS only; used in PS1 to show whether REF to 9DBINIT or 9INITIAL set; see INIT2 and WRT; also used to turn off trigger.

NAME	DISPLACEMENT	CONTENTS
BFR	+175	Pointer into BUF; storage for checksum in GBYTE; also in FIELDEX routine of EVL used in switching logic for define field.
FIRSTF	+176	Pointer into REF/DEF stack for first forward reference.
LASTF	+177	Pointer into REF/DEF stack for last forward reference.
XCSEG1	+178	See XMEM logic of EVL; retains current segment to permit alternate use of CSEG1 for XMEM.
SYMBOLTB	+179	Base of current internal symbol table.
SYMTOP	+180	Top of current internal symbol table.
TRAPCC	+181	For UTS; retains condition codes when loader enters its trap handler for trap 40; see LDR.
CODE	+182	New field of information output with diagnostics; part of QUIT procedure.

Footnotes for Figure 10:

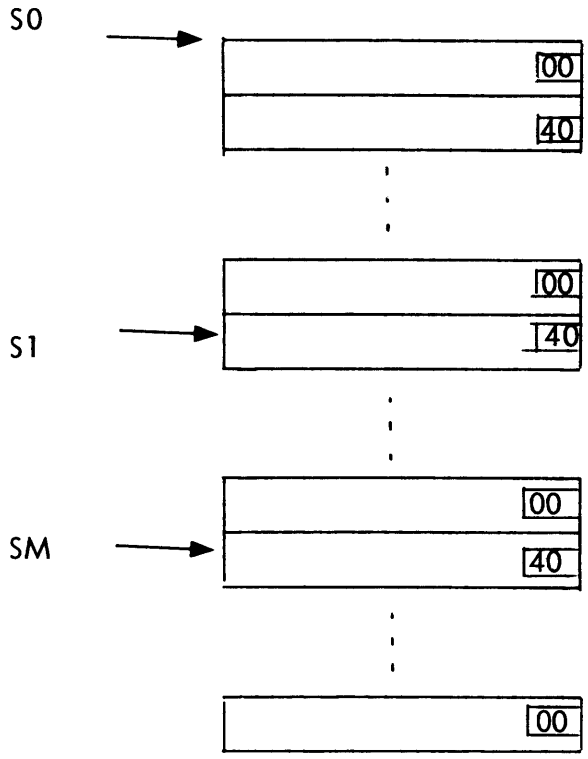
*NOTE:

A =	1, MAP specified	K =	1, GO specified
B =	1, NOSYSLIB specified	L =	1, BI specified
C =	1, REF specified	M =	1, CSEC1 was specified
D =	1, PERM specified	N =	1, NOTCB specified
E =	1, LIB specified	O =	1, XMEM in effect (set by the Loader in IN2)
F =	1, M10 specified		
G =	1, M100 specified	P =	1, MAP, VALUE specified
H =	1, FCOM specified	=	0, MAP, NAME specified
I =	1, ABS specified	Q =	1, BREF specified
J =	1, Assigns Read	R =	1, EF specified
SL =	Sev. Level (default = 4)		

**BPM-UTS differences in the LOCCT Tables:

	Word	BPM	UTS
	4	LOAD BIAS field Default = 0	LOAD BIAS Default = background lower limit WA
	5	Background lower limit	------
***	6	Passed to the Loader in Reg. D4 (D4) = FCOM size	

The ROM Tables contain an entry for every input file (ROM or load module). Below is an overall picture for M segments (S0, S1, ... SM). Each box is a seven-word entry.



ROM Table Entry

0	TEXTC of ROM name (EF name, or SYSid B
1	for BI, or SYSid G for GO).
2	0 X L 0 0 0 T 0
3	Account number
4	(default = current acct#)
5	Password
6	(default = 0, 0)

X = 1, if this is not the last ROM file in the segment.
L, T are initially 0 but set by the Loader.
L = 1, if file came from a library.
T = 1, if file is on labeled tape.

Figure 11. ROM Tables

XEROX Publication Revision Sheet

August 1972

CORRECTIONS TO XEROX BTM/BPM/UTS OVERLAY LOADER TECHNICAL MANUAL (Sigma 5-9 Computers)

PUBLICATION NO. 90 18 03B, JUNE 1972

The attached pages contain changes that update the manual to the C00 version of UTS and the G00 version of BPM/BTM. Pages in the B edition of the manual that are to be replaced are: iii, iv, 5, 6, 17-20, 23, 24, 27-30, 39-42, 55, 56, 61, 62, 67-72, 87-90, 113-118. New pages to be inserted are: 133-139.

These changes will be incorporated into the next edition.

Revision bars in the margins of replacement pages identify changes. Pages without the publication number 90 18 03B-1(8/72) at the bottom are included only as backup pages; revision bars appearing on such pages identify changes made in a previous revision.