CRAY RESEARCH, INC.

# CRAY X-MP AND CRAY-1® COMPUTER SYSTEMS

## SEGMENT LOADER (SEGLDR) REFERENCE MANUAL

SR-0066

RECORD OF REVISION


PUBLICATION NUMBER SR-0066

Each time this manual is revised and reprinted, all changes issued against the previous version in the form of change packets are incorporated into the new version and the new version is assigned an alphabetic level. Between reprints, changes may be issued against the current version in the form of change packets. Each change packet is assigned a numeric designator, starting with 01 for the first change packet of each revision level.

Every page changed by a reprint or by a change packet has the revision level and change packet number in the lower righthand corner. Changes to part of a page are noted by a change bar along the margin of the page. A change bar in the margin opposite the page number indicates that the entire page is new; a dot in the same place indicates that information has been moved from one page to another, but has not otherwise changed.

Requests for copies of Cray Research, Inc. publications and comments about these publications should be directed to:

CRAY RESEARCH, INC.,
1440 Northland Drive,
Mendota Heights, Minnesota 55120

| Revision | Description |
|---|---|
| | September, 1983. Original printing. |
| 01 | December, 1983. This change packet brings the manual into agreement with version 1.13 of COS. New material includes the ALIGN, HEAP, and STACK directives, and related error messages. |
| A | November, 1984. This reprint with revision brings the manual into agreement with version 1.14 of COS. New material includes the MLEVEL, LOWHEAP, and SID directives, and related error messages. All previous versions are obsolete. |

# PREFACE

This manual introduces SEGLDR, a Cray Research, Inc., automatic loader for overlayed or nonoverlayed programs. The manual assumes familiarity with FORTRAN programs, the function of a loader, and a general understanding of overlays.

Intended for reference, the manual describes SEGLDR operation and program segmentation. Appended to this manual is a glossary of terms you will need to understand in the context of the use and operation of SEGLDR.

For information on SEGLDR tables and subroutines, see the COS Products Set Internal Reference Manual, publication SM-0041. SEGLDR error messages are described in Appendix F of this manual and in the CRAY-OS Message Manual, publication SR-0039.

# CONTENTS

APPENDIX SECTION



FIGURES

FIGURES (continued)



TABLE

# INTRODUCTION 1

SEGLDR is an automatic loader for code produced by language processors such as CAL (Cray Assembly Language) or CFT (Cray FORTRAN). Program segments (described in section 2) are loaded as required without explicit calls to an overlay manager.

In this publication, overlayed codes are termed *segmented* programs and nonoverlayed codes are termed *nonsegmented*. Executing under the control of the Cray Operating System (COS) on all Cray Computer Systems, SEGLDR can produce segmented or nonsegmented object modules (executable binary programs).

With SEGLDR, segmented programs can be produced and executed without extensive user code modification.

- Since you specify the segment structure and contents, SEGLDR can detect subroutine calls that require loading new segments into memory.
- A resident routine loaded with the object module handles program overlay management.

In addition to automatic segment loading and unloading, major advantages of SEGLDR include the following.

- You can easily modify code overlay structure by changing SEGLDR directives, usually without recompilation.
- By altering the input directives to SEGLDR, you can experiment, overlaying different code without making significant source code changes.
- Normally you need not specify more than one module (subroutine) per segment for SEGLDR to assign all contents to a segment.
- You need not group modules for each segment together in a single dataset for loading. The modules can be scattered among several binary datasets and libraries.
- SEGLDR can pass arguments between subprograms residing in different segments.
- SEGLDR can unload segments and any common blocks they contain and subsequently reload them with their updated image retained.

For both segmented and nonsegmented programs, SEGLDR provides equivalencing of entry point names and control of the common block loading order. (See the glossary section for a definition of *entry point* as it is used throughout this manual.)

In addition, a common block other than blank common can be specified as dynamic.

SEGLDR is called into execution using a control statement in a job input dataset. Job input datasets and the SEGLDR control statement are described in section 3.

The features provided by SEGLDR are independent of overlay facilities provided by LDR (see the CRAY-OS Version 1 Reference Manual, publication SR-0011).

# PROGRAM SEGMENTATION 2

With SEGLDR, you specify the segment structure and the content of the segments to be loaded. This section describes the principles of SEGLDR program segmentation, or *tree* design (irrelevant for nonsegmented programs). Section 3 describes SEGLDR input and output, and section 4 describes directives for specifying *tree* shape and segment contents.

## SEGLDR SEGMENT TREE CONCEPT

With SEGLDR, program segments are arranged in a tree structure, as illustrated in figure 2-1. (Note that a nonsegmented program consists of only one segment, the root segment.)

Each segment in a tree contains one or more subprogram modules, and possibly some common blocks. Subprogram hierarchy helps you determine the shape of your tree.

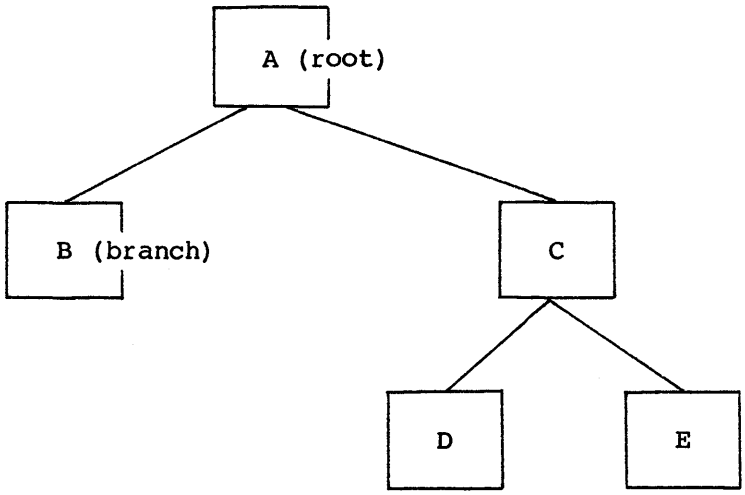



Figure 2-1. A segment tree

Each segment in the preceding figure is assigned an arbitrary but unique 1- to 8-character segment name.

The apex of the SEGLDR segment tree (segment A in figure 2-1) is called the *root* segment. The remaining segments, B, C, D, and E, are called *branch* segments. Within these branch segments, B, C, D, and E are referred to as *successor* segments of A. B and C are called *immediate successors* of segment A, and segments D and E are immediate successors of segment C. It follows, then, that C and A are *predecessor* segments for D and E, and A by itself is the predecessor segment for B and C. C is the *immediate predecessor* of segments D and E. Note that the root segment is a predecessor for every branch segment and has no predecessor segment itself. Predecessor and successor segments lie on a common branch. Down the tree (or branch) is moving away from the root segment, and up is moving toward it.

A segment *level* is the number of immediate successor segments that must be traversed when proceeding from the root segment to the destination segment. For example, the root segment is level 0, segments B and C are level 1, and segments D and E are level 2.

During program execution, only one segment from each level can be in memory at a time. The root segment is always memory resident; other segments occupy higher memory addresses when required. In general, predecessor segments of the executing segment are guaranteed to be memory resident. In addition, successor segments at higher levels might be memory resident, depending on recent subroutine calls to successor segments.

## SEGLDR SEGMENT TREE DESIGN

The only restriction on the height or width of the segment tree is that a maximum of 1000 segments, including the root, can be defined. However, you must adhere to the following rules for a segment tree to be valid.

- Each segment tree can have only one root segment (a segment with no predecessor segments) and must have at least one branch segment.
- Each nonroot segment must have only one immediate predecessor segment.

Figures 2-2 and 2-3 illustrate valid segment trees.

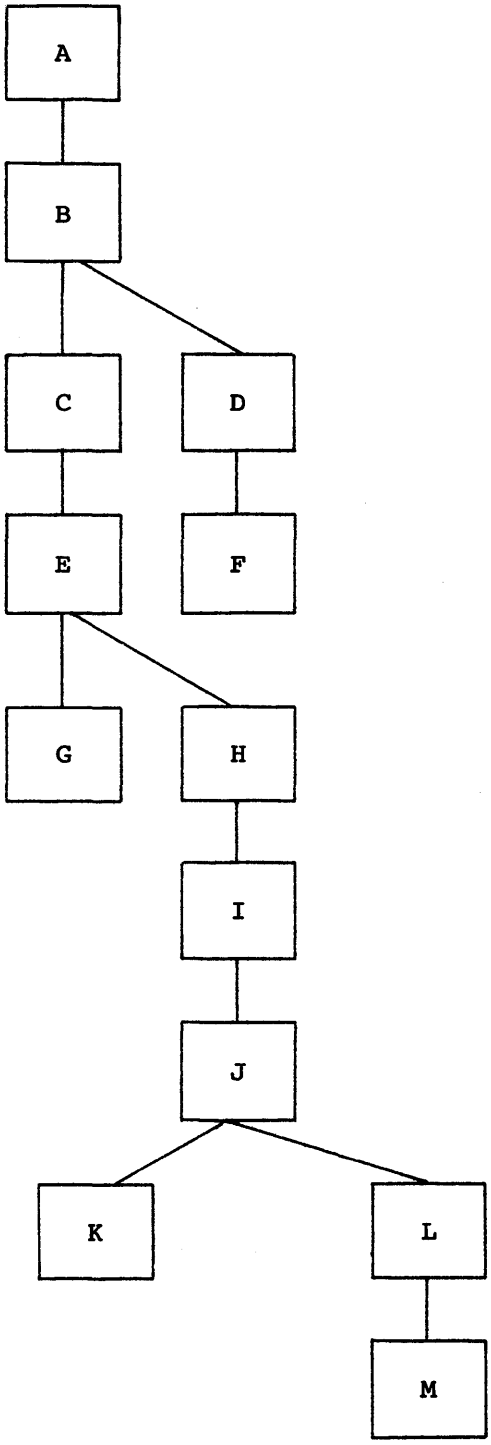



Figure 2-2. Valid segment tree

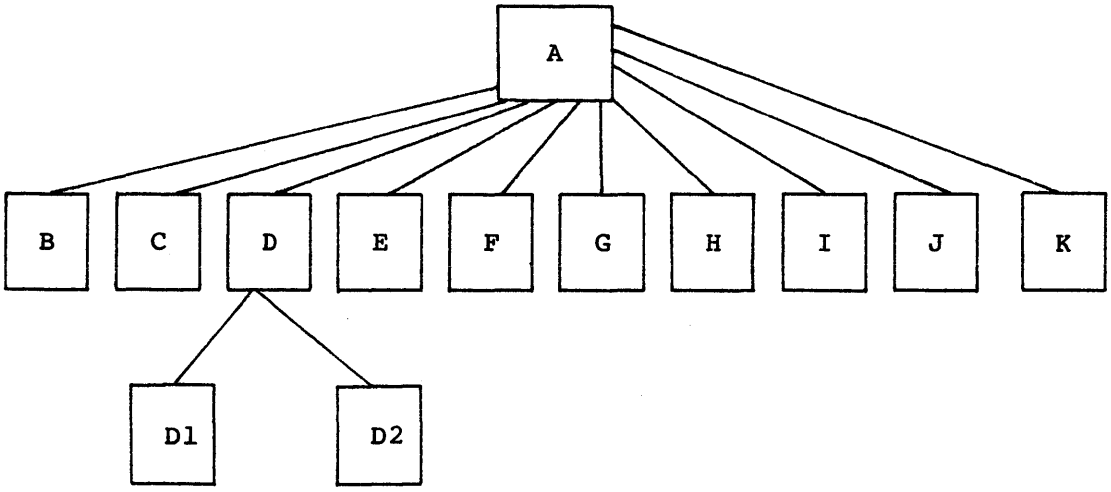



Figure 2-3. Valid segment tree

Figures 2-4 and 2-5 show tree structures that are invalid because of their multiple root segments or multiple immediate predecessor segments.

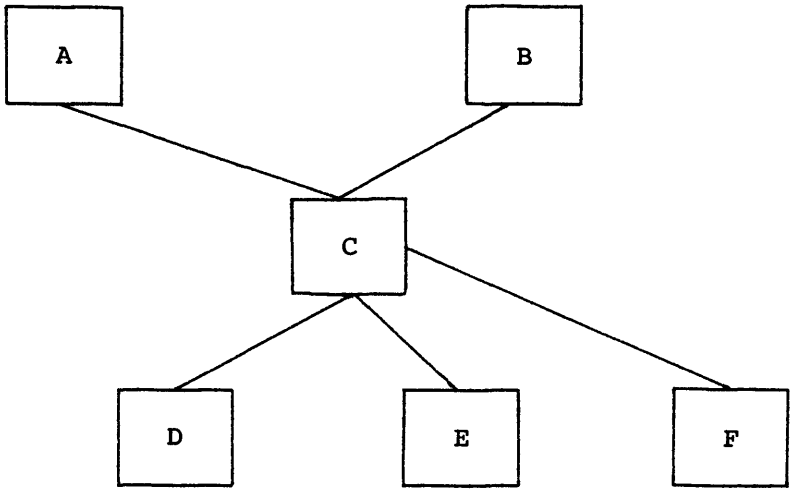



Figure 2-4. Invalid segment tree (multiple root segments)

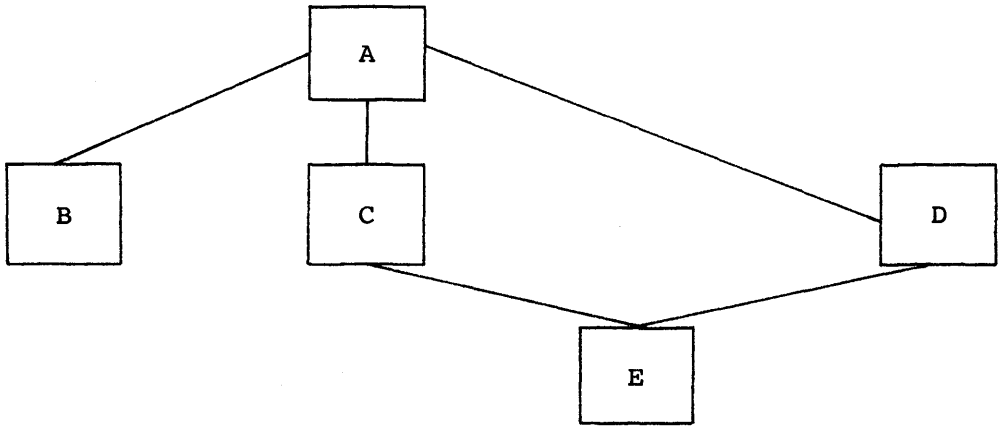



Figure 2-5. Invalid segment tree (multiple immediate predecessor segments)

## SUBROUTINE CALLING BETWEEN SEGMENTS

Calls can be made from any module (subroutine or function) in a segment to any module in a successor or predecessor segment. Calls across the segment tree are illegal. That is, subroutine calls can be made both up and down the tree, as long as the calling and called modules are owned by segments on a common branch. If a call is made to a subroutine in a non-immediate successor segment, all segments on the branch are read to memory. (See section 5 and Appendix A for special rules affecting duplicate modules and common blocks.)

When a call is made from a subroutine to a subroutine further down the branch at execution time, SEGLDR intercepts the call, loads the appropriate segment or segments if not already in memory, passes the arguments, and jumps to the called entry point (see *entry point* in the glossary). SEGLDR intercepts only the calls to subroutines in successor segments, because they are the only calls that could cause a segment to be loaded (all callers of a segment in memory are already in memory).

**********

CAUTION

In CAL, use of the CALL and CALLV macros is strongly recommended for subroutines calls to other modules.

**********

Do not pass an entry point to a subroutine as an argument if the entry point is not in the same or a predecessor segment. For example:

```
EXTERNAL JOE
 CALL SUB (JOE)
```

The external reference cannot be detected by SEGLDR at load time and may not be in memory.

Do not expand memory at the end of a segment that has one or more successor segments because, if a successor segment is in memory, it is overwritten; if a successor segment is not in memory but is brought in later, it overwrites the expanded area. Use of dynamic common blocks is recommended instead. (See the DYNAMIC directive in section 4.)

SEGLDR handles subroutine calls as shown in the figure 2-6. The numbers 1 through 5 represent modules in segments A through E.

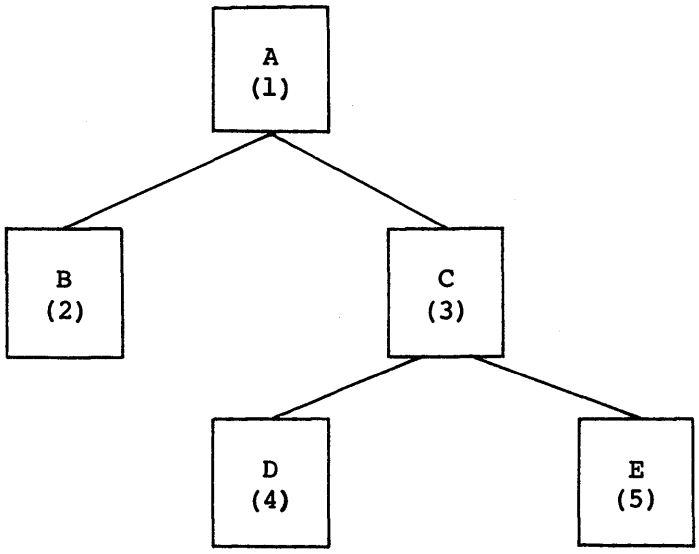



Figure 2-6. Subroutine handling

The subroutine call descriptions following are related to the tree structure shown in figure 2-6.

| From | To | Comment |
|---|---|---|
| 1(A) | 2,3,4,5 | Legal; may need to load some segments. |
| 2 | 1 | Legal; no load needed. |
| 2 | 3,4,5 | Illegal; calls across a branch. |
| 3 | 2 | Illegal; calls across a branch. |
| 3 | 1,4,5 | Legal; may need to load if to modules 4 or 5. |
| 4 | 5,2 | Illegal; calls across a branch. |
| 4 | 1,3 | Legal; no load needed. |
| 5 | 4,2 | Illegal; calls across a branch. |
| 5 | 3,1 | Legal; no load needed. |

# SEGLDR JOB FLOW 3

SEGLDR constructs object modules (executable binary programs) in two phases:

- Input analysis
- Code construction on a segment-by-segment basis

At the conclusion of the first phase, SEGLDR has identified all blocks required for loading and knows their sources (load datasets) and destinations (segments). Error discovery, excluding field relocation overflow errors, is part of the first phase.

During the second phase, address relocation and data loading (see the glossary at the back of this manual) are performed. At the conclusion of this phase, mapping options are honored. For more information on SEGLDR map options, see the MAP directive in section 4.

Figure 3-1 depicts SEGLDR data flow.

## INPUT TO SEGLDR

Input to SEGLDR consists of:

- Directives controlling construction of the object module. (See section 4 for descriptions and Appendix E for an example of putting the directives together.)
- Relocatable *binary input* (BIN) and *library datasets* from language processors, such as Cray Assembly Language (CAL) or Cray FORTRAN (CFT). A binary input dataset consists of one record for each module (subroutine or function) compiled or assembled.

SEGLDR recognizes three types of binary input datasets. The three types of binary input datasets are identical; they differ only in their use by SEGLDR. The three types are the following.

- Global BIN datasets
- Segment description BIN datasets
- Library (LIB) datasets

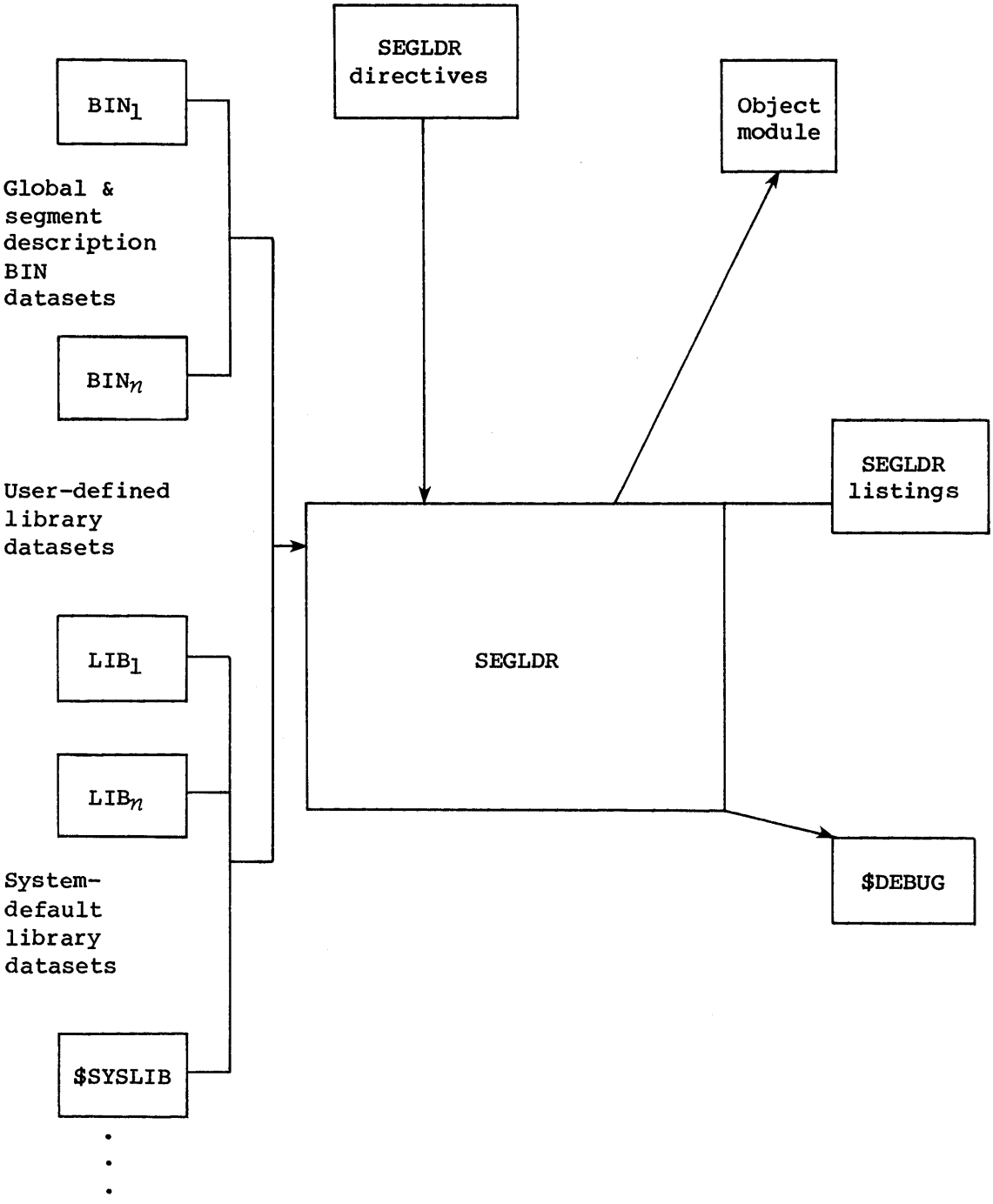



Figure 3-1. Data flow

## GLOBAL BIN DATASETS

A global BIN dataset must consist of at least one relocatable binary record produced by language processors such as CAL, or CFT. Global BIN datasets are defined by the global BIN directive as containing the modules to be loaded (see section 4).

## SEGMENT DESCRIPTION BIN DATASETS

Modules in segment description BIN datasets and global BIN datasets are the same, except that modules in segment description BIN datasets are assigned to specific segments. The segments to which they are assigned are defined by the segment description BIN directive (see section 4).

## LIBRARY (LIB) DATASETS

Like the global and segment description BIN datasets, datasets named by a LIB directive must contain at least one relocatable binary record produced by language processors such as CAL, or CFT.

Datasets defined by the LIB directive contain modules to be used in resolving references to unknown entry points called *unsatisfied externals*. When all BIN datasets have been examined, any remaining unsatisfied externals are resolved using LIB datasets, if possible. In resolving unsatisfied externals, SEGLDR examines the LIB datasets in the order specified by the LIB directive (see section 4). Not all LIB datasets are always examined, since SEGLDR stops scanning LIB datasets once all unsatisfied externals have been resolved.

In the event that an entry point appears in more than one library, the first occurrence of a relocatable binary record that will satisfy the external reference is used. Because SEGLDR maintains all pertinent information about all of the libraries simultaneously, it can resolve unsatisfied externals without rescanning any libraries.

## DATASET PROCESSING

SEGLDR examines only the first file of binary input datasets. All datasets must be local to the job or must reside in the system directory (SDR). A dataset local to the job is used in preference to an identically named dataset in the system directory.

Modules need not be grouped in a single dataset for each segment in order to be loaded. They can be scattered among several binary input datasets.

All modules within binary input datasets are initially assumed to be required for the load. SEGLDR gathers the entry points of all modules in all datasets specified with global BIN and LIB directives, and then discards all modules that are never called†, except for the module containing the load transfer entry point and BLOCKDATA subprograms. See the FORTRAN (CFT) Reference Manual, CRI publication SR-0009, for further information on BLOCKDATA.

## OUTPUT FROM SEGLDR

Output from SEGLDR is:

- The object module
- Listing output
- Symbol table dataset

The object module (the executable binary program produced by SEGDLR) is formatted so that it can be loaded by COS for execution. Segmented codes have one record per segment. The root segment is written to the first record. Within the root is a resident routine (see section 5) that handles intersegment subroutine calls.

SEGLDR writes the object module to the first file of the ABS dataset (see the ABS directive in section 4). For nonsegmented programs, only the first record of the ABS dataset is used.

SEGLDR can echo all input directives to the listing dataset. (Appendix E contains an example input directive listing.) A block map supplied by the MAP directive (see section 4), an entry-point cross-reference, a common block/module reference and a summary of unsatisfied externals can also be requested. Appendix G contains map examples. The user has control over the severity of error messages that should be written to the listing dataset.

SEGLDR generates a symbol table dataset suitable for input to SID, the CRI symbolic interactive debugger. For more information, see the Symbolic Interactive Debugger (SID) User's Guide, CRI publication SG-0056.

† The process of discarding modules is refered to as *tree trimming* in this manual.

## SEGLDR CONTROL STATEMENT

Execute SEGLDR with the following control statement.

Format:

```
SEGLDR,I=idn,L=ldn,DW=dw,CMD='dirstr'.
```

Parameters:

I=*idn* Name of input dataset containing SEGLDR directives. If you omit this parameter, I=0, indicating there are no input directives. If I is specified without *idn*, *idn* assumes its default value: $IN.

L=*ldn* Name of dataset for printable output. If you omit this parameter or if you specify L without *ldn*, *ldn*=$OUT. L=0 suppresses all listable output (including error messages).

DW=*dw* Data width for input directives; that is, the number of significant columns in each input line. DW=72, for example, allows SEGLDR to ignore UPDATE sequence numbers (columns 73-96). (For more information on UPDATE see the UPDATE Reference Manual, CRI publication SR-0013.) If you omit this parameter or you specify it as DW or DW= only, SEGLDR assumes DW=80. The data width must be in the range 0< *width* <81.

CMD='*dirstr*'
Global directives to be processed by SEGLDR as if the string is the first image read from the input dataset (I=*idn*). Separate the directives with semicolons. The directive string is processed even if you specify I=0.

Example:

```
SEGLDR,CMD='BIN=BIN1,BIN2;LIB=MYLIB;MAP=PART'.
```

# SEGLDR DIRECTIVES 4

Three types of directives convey information to SEGLDR:

- Global directives
- Segment tree definition directives
- Segment description directives

Global directives identify binary datasets to be loaded and select control options. Segment tree definition directives convey the tree shape. Segment description directives specify the contents of individual segments that make up the tree. Appendix E includes an example of all three directives types in use together.

Global directives apply to both segmented and nonsegmented codes. For nonsegmented loading you can only specify global directives. For segmented loading you must specify both segment tree definition and segment description directives.

In this section, the three types of directives are discussed separately and arranged alphabetically by type. Only the global directives are broken into groups according to subtype. Error messages related to directive use are included with the other SEGLDR error messages in Appendix F. See the glossary for a definition of *entry point* as it is used throughout this section.

## SEGLDR SYNTAX

Most SEGLDR directives have KEYWORD=*value* syntax. Exceptions are stated in individual directive descriptions. The following paragraphs describe the conventions used in representing SEGLDR directives and the actual syntax of SEGLDR directives.

## CONVENTIONS

This manual uses the following conventions.

- Variable names are represented in italics; actual names are shown in uppercase.
- Spaces are insignificant and are used in this manual only for clarity.
- Default values are underlined.
- Brackets [] enclose a list of optional elements.
- Braces {} enclose two or more elements when one of them must be chosen.

## DIRECTIVE SYNTAX

The directive syntax is as follows:

- SEGLDR directives can be entered as uppercase, lowercase or mixed-case input, since SEGLDR converts the directive string to uppercase before evaluation.
- Comments are preceded by an asterisk and can appear anywhere in the input. All characters to the right of an asterisk are ignored.
- Each directive is terminated by a semicolon, an asterisk, or an end of line.
- Multiple directives on a single line are individually terminated with a semicolon.
- Elements in a list are separated by commas.
- SEGLDR ignores null directives (for example, two successive semicolons).
- You can continue some SEGLDR directives on following lines. These directives have a comma as the last nonblank character before the end of line. See individual directive descriptions for more detail.

## GLOBAL DIRECTIVES

Global directives identify binary datasets to be loaded and select various control options. Global directives can be entered in any order. The global directives are grouped in this manual according to function: listing, input, entry point control, data description, security, memory management, and miscellaneous. Note that all the directives included in the examples are described in this section.

### GlOBAL LISTING DIRECTIVES

The following global directives control and provide options for listed output.

- COMMENT
- ECHO
- MAP
- MLEVEL
- TITLE

COMMENT, ECHO, and TITLE directives can be included as either global or segment description directives.

#### COMMENT directive

The COMMENT directive, which annotates SEGLDR directives, is echoed to the listing dataset but is otherwise ignored. All characters to the right of the asterisk are part of the comment string.

Continuation beyond one line is not allowed.

You can use the COMMENT directive in either the global or the segment description directives portion of the input.

Format:

```
* comment string
```

Example:

```
TITLE=GLOBAL DIRECTIVES
**********************
*  Global directives
**********************
BIN=X
TITLE=TREE DIRECTIVES
**********************
*
*Tree directives
**********************
TREE
   ROOT(A,B)
ENDTREE
TITLE=SEG.DESCR.DIR.
**********************
SEGMENT=ROOT
```

## ECHO directive

The ECHO directive resumes or suppresses printing of input directives. You can use it in either the global or the segment description directives portion of the input. If you do not use the ECHO directive, ECHO=OFF.

Format:

```
ECHO={ON  }
     {OFF }
```

Parameters:

| | |
|---|---|
| ON | Resumes listing of input directives |
| OFF | Suppresses directive listing. The use of ECHO=OFF does not prevent printing of error diagnostics. SEGLDR automatically echoes erroneous directive lines followed by an error message. |

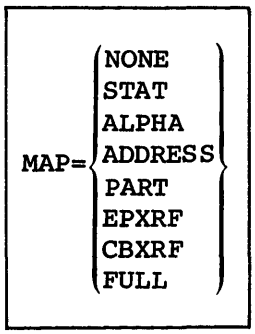
ECHO has no effect if L=0 is specified on the SEGLDR control statement.

You may not continue this directive on a second line.

## MAP directive

The MAP directive controls SEGLDR map output generation. Besides memory mapping, the MAP directive provides time and date of load, length of longest branch and last segment, and transfer address. Map output is written to the listing dataset. See the examples in Appendix D.

Format:




Parameters:

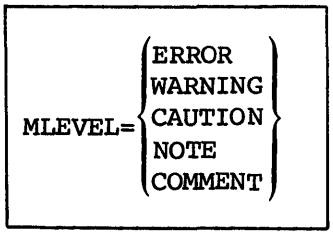
| | |
|---|---|
| NONE | Writes no map output to the listing file. If you do not use the MAP directive, MAP=NONE. |
| STAT | Writes statistics for the load such as date and time, length of longest branch, last segment, transfer entry point, and stack and heap information. |
| ALPHA | Writes the STAT information plus the block map for each segment, listing the modules in alphabetical order. |
| ADDRESS | Writes the ALPHA information but lists modules by ascending load address. |
| PART | Writes both ALPHA and ADDRESS information. |
| EPXRF | Writes the Entry Point Cross Reference Table. |
| CBXRF | Writes the Common Block Cross Reference Table. |
| FULL | Writes all PART, EPXRF, and CBXRF information. |

You may not continue this directive on a second line.

MLEVEL directive

The MLEVEL directive controls SEGLDR message printing on the listing output. The keyword indicates the lowest priority error message to be printed. If you do not use the MLEVEL directive, MLEVEL=CAUTION.

Format:




Parameters:

ERROR — Prints only the most severe error messages. This level of severity immediately terminates SEGLDR and no executable output is written. You can suppress ERROR level messages by specifying L=0 on the SEGLDR control statement.

WARNING — Prints ERROR and WARNING levels of error messages. A WARNING level message indicates that the executable output is not written but proccessing continues so that additional messages may be printed.

CAUTION — Prints ERROR, WARNING, and CAUTION levels of error messages. A CAUTION level message indicates that an error possibly occurred, but is not severe enough to prohibit generation of executable output.

NOTE — Prints ERROR, WARNING, CAUTION, and NOTE levels of error messages. A NOTE level message indicates that SEGLDR may have been misused or used inefficiently. This level of messages has no effect on execution validity.

COMMENT — Prints all levels of error messages. A COMMENT level error message does not effect execution.

MLEVEL has no effect if you specify L=0 on the SEGLDR control statement.

You cannot continue this directive on a second line.

### TITLE directive

The TITLE directive places an arbitrary, user-defined character string in the second line of each page header. A page eject is forced so that following directive records are written to a new page. The title line is initially clear and can be reset by TITLE directives in either the global or the segment description directives portion of the input. If you specify the directive as TITLE or TITLE= only, the title line is cleared.

Format:

```
TITLE[=title string]
```

Parameter:

*title string*
: User-defined character string; maximum length is 74 characters.

You cannot continue this directive on a second line.

Example:

```
TITLE=Place this in the page header, please.
```

The TITLE directive copies the string "Place this in the page header, please." verbatim to the page header. It performs no character editing (for example, blank suppression or uppercase shifting). An end-of-record or a semicolon signals the end-of-the-title string.

## GLOBAL INPUT DIRECTIVES

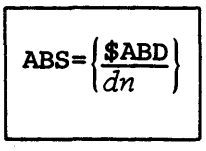
The following global directives provide dataset information to SEGLDR.

- ABS
- BIN
- LIB
- NODEFLIB

## ABS directive

ABS specifies the dataset to receive the object module constructed by SEGLDR. If you do not use the ABS directive, SEGLDR assumes dataset name $ABD.

Format:




Parameter:

*dn* Names the dataset to receive the object module

The ABS dataset is rewound before and after being written.

You cannot continue this directive on a second line.

## BIN directive (global)

The global BIN directive names binary input datasets to be searched. Only the first file of each dataset is processed. Remaining files are ignored without comment.

The effect of multiple global BIN directives is cumulative.

The global BIN directive functions differently from the segment description BIN directive in that modules appearing in global BIN datasets are not assigned to a specific segment. Using both BIN and MODULES directives to name the same module causes a fatal error.

If the directive lists multiple binary input datasets, SEGLDR proccesses them in the order specified. Consequently, if an entry point is present in more than one dataset, SEGLDR loads the first module encountered containing the entry point. Note that if you use the MODULES directive, this rule may not apply. That is, it is not true for modules named by the MODULES directive that specify an input dataset.

SEGLDR assumes that all modules within global BIN datasets are movable (not assigned to any segment) and that, initially, all modules are required in the load. After SEGLDR examines all binary files and libraries, it discards the modules that are never called (unless you specify the FORCE directive option ON). The only exceptions are the module containing the initial transfer address and BLOCKDATA subprograms.

Format:

BIN=$dn_1$,$dn_2$,$dn_3$,...,$dn_n$

Parameters:

$dn_i$ Names of binary input datasets to be loaded. If no dataset is named by a global BIN directive, the default is $BLD.

If you continue this directive beyond one line, end each line to be continued with a comma.

Example:

```
BIN=JOE,SALLY,HARRY,
    WILLIAM
```

LIB directive

The LIB directive is used to augment the default list of libraries for the load. Library datasets specified with the LIB directive are searched before any default libraries.

The effect of multiple LIB directives is cumulative.

This directive is the same as the BIN directive, except that only previously *unsatisfied externals* are loaded. An unsatisfied external is a reference (for example, a subroutine call) to an unknown entry point.

Format:

LIB=$lib_1$,$lib_2$,$lib_3$,...$lib_n$

Parameters:

$lib_i$ Names of libraries you provide

Example:

The following example defines seven user libraries to be searched before default libraries when SEGLDR processes subprogram linkages (matches callees with callers). The search order is LIB1, LIB2,... LIB6, LIB7.

```
LIB=LIB1,LIB2,LIB3,LIB4,
    LIB5
LIB=LIB6,LIB7
```

If you continue this directive beyond one line, end each line to be continued with a comma.

NODEFLIB directive

NODEFLIB instructs SEGLDR to ignore all default libraries. Only modules found in datasets declared by BIN and LIB directives are considered for loading. With NODEFLIB, you are responsible for providing all modules required for code execution. For a segmented load, include $SEGRES, the SEGLDR runtime resident routine (see section 6).

Format:

```
NODEFLIB
```

You cannot continue this directive on a second line.

Example:

The following example tells SEGLDR to search libraries MYLIB1, MYLIB2, $ARLIB, $MYSYS, and $MYSCI to match callers with callees. SEGLDR does not revert to default libraries for entry points located in unspecified libraries.

```
NODEFLIB; LIB=MYLIB1,MYLIB2,
          $ARLIB,$MYSYS,
          $MYSCI
```

GLOBAL ENTRY POINT CONTROL DIRECTIVES

The following global directives name entry points.

- EQUIV
- MODULES
- USX
- XFER

EQUIV directive

By assigning synonyms to an entry point name, EQUIV substitutes a call to one entry point for a call to another.

Format:

EQUIV=*epname*(*syn*$_1$, *syn*$_2$..., *syn*$_n$)

Parameters:

*epname* Names a target entry point

*syn*$_i$ Names entry points to be linked to *epname*

If you continue this directive beyond one line, end each line to be continued with a comma.

Example:

Consider the following code sequence.

```
      .
      .
      .
   CALL A
      .
      .
      .
   CALL B
      .
      .
      .
```

The calls to A and B are linked to C as follows:

```
   EQUIV=C(A,B)
```

Note that the module containing entry point C is loaded, but the module or modules containing A and B may not be loaded. The module or modules containing A and B may be loaded if needed to satisfy other references to other entry points. The process is similar to using a text editor to replace all occurrences of CALL A and CALL B with CALL C.

MODULES directive (global)

The MODULES directive names modules to be loaded. The global MODULES directive specifies the dataset from which to obtain a module if modules of the same name are in different datasets.

Format:

MODULES=*modname*$_1$:*dsname*$_1$,*modname*$_2$:*dsname*$_2$,...*modname*$_n$:*dsname*$_n$

Parameters:

*modname*$_i$ Name of module to be loaded

*dsname*$_i$ Name of the dataset from which to obtain the module

Example:

```
MODULES=SUBB:LIB1,SUBD:DTASET1
```

In this example, the MODULES directive obtains SUBB from library LIB1 and SUBD from binary input dataset DTASET1.

USX directive

USX controls whether unsatisfied external symbols are treated as loading errors. If you do not use the USX directive, USX=CAUTION.

Format:

USX={WARNING / CAUTION / IGNORE}

Parameters:

WARNING SEGLDR treats an unsatisfied external symbol as a warning message and does not write executable output.

CAUTION SEGLDR treats an unsatisfied external symbol as a caution message and writes the executable output.

IGNORE If SEGLDR encounters an unsatisfied external symbol, it writes the executable output but does not write the message.

You cannot continue this directive on a second line.

XFER directive

The XFER directive names the entry point SEGLDR transfers control to when execution begins. If you do not use the XFER directive SEGLDR uses the first primary entry point discovered as the transfer entry point.

Format:

```
XFER=entry
```

Parameter:

*entry* Entry point name

You cannot continue this directive on a second line.

GLOBAL DATA DESCRIPTION DIRECTIVES

The following global directives describe data handling.

- COMMONS
- DYNAMIC
- PRESET
- SLT

COMMONS directive (global)

The global COMMONS directive causes the listed common blocks to be loaded in the indicated order. However, with a segmented load, the global form of this directive has no effect.

Format:

```
COMMONS=comblk1,comblk2,...comblkn
```

Parameter:

*comblk*$_i$ Names the common blocks to be loaded and specifies the loading order

If you continue this directive beyond one line, end each line to be continued with a comma.

DYNAMIC directive

The common block named by the DYNAMIC directive occupies memory following the largest code segment. (And after the heap if it is present.) The common block can expand or contract under user control.

All segments have access to dynamic common at any time during program execution.

The dynamic common block program space is not physically allocated during code construction by SEGLDR, and so may not be data loaded (see the glossary). All references to variables in the dynamic common block, however, are properly relocated.

Format:

DYNAMIC={*comblk* | //}

Parameters:

*comblk* Relocates named common block to the first word following the longest segment branch. Only one common block can be named.

// Specifies blank common as dynamic

There is no default dynamic common block.

You cannot continue this directive on a second line.

The ORDER directive has no effect on DYNAMIC common block placement. For example, ORDER=LBC; DYNAMIC=SPACE places all labeled common blocks except /SPACE/ first, then blank common, then all code blocks, and finally /SPACE/.

Example:

CFT program

```
PROGRAM X
COMMON /DYNCOM/ SPACE(1)
.
.                    User requests 9999 additional words of memory.
.                    COS adds memory to the end of SPACE array.
.
DO 100 I=1,10000
    SPACE(I)=0       Zeros out 10,000 words, but only one word is
                     actually pre-allocated by SEGLDR.
100 CONTINUE
```

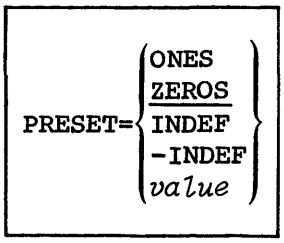
SEGLDR directive

DYNAMIC=DYNCOM identifies /DYNCOM/ as the dynamic common block.

PRESET directive

The PRESET directive specifies a value that SEGLDR uses to preset uninitialized data areas within the object module (for example, variables in labeled common blocks with no DATA statements). If you do not use the PRESET directive, PRESET=ZEROS.

Format:




Parameters:

| | |
|---|---|
| ZEROS | Sets uninitialized data to 0 (default) |
| ONES | Sets uninitialized data to -1 |
| INDEF | Sets uninitialized data to a value that generates a floating-point error if used as an operand in a floating-point operation $(06050540000000000000)_8$ |

-INDEF Same as INDEF except the preset value is negative ($1605054000000000000)_8$

*value* Inserts a 16-bit octal value into each parcel of uninitialized data, where $0 < bits < 177777_8$

You cannot continue this directive on a second line.

## SLT directive

The SLT directive specifies the size of the Segment Linkage Table (SLT). SEGLDR writes the SLT to the root segment for servicing intersegment subroutine calls. SEGLDR writes the actual SLT requirement to the listing dataset upon load completion. If SLT specifies a size less than the actual requirement, an error message specifies the actual requirement.

Format:

```
SLT=nnn
```

Parameter:

*nnn* Size (decimal word count) to be reserved for the Segment Linkage Table

By default, SEGDLR computes the size of the SLT according to the following formula: SLT=40*NBRNCH, where NBRNCH is the number of nonterminal segments (segments having at least one successor segment). This formula allows for an average maximum of 40 intersegment subroutine calls to successor segments. Calls to predecessor segments need no resident loader intervention.

You cannot continue this directive on a second line.

# GLOBAL SECURITY DIRECTIVES

Global security directives are the following.

- GRANT
- SECURE

## GRANT directive

This directive indicates which privileges to grant when SEGLDR loads the absolute module (the dataset specified by the ABS directive) from the System Directory (SDR). You may specify any or all of the parameters in a single GRANT directive. In producing the absolute module, SEGLDR merges these privileges with existing privileges.

Format:

GRANT=$privilege_1$,$privilege_2$,...$privilege_n$

Parameters:

$privilege_i$

| | |
|---|---|
| SCRDSC | Read DSC/DXT page |
| SCSPOL | SAVE/ACCESS/DELETE/LOAD/DUMP/spooled dataset |
| SCLUSR | Load user dataset |
| SCDTIM | Dump time request |
| SCQSDT | Dequeue/queue SDT requests |
| SCUPDD | Access user dataset for PDSDUMP |
| SCACES | Access user-saved dataset without passwords |
| SCQDXT | LINK/MODIFY DXT requests |
| SCENTR | ENTER option on ACCESS |
| SCNVOK | Invoke job class structure |
| SCDUMP | Allow F$DJA requests any time |
| SCPRIV | Allow special system requests |

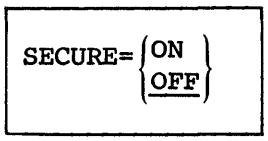
If you continue this directive beyond one line, end each line to be continued with a comma.

These privileges are discussed in the CRAY-OS Version 1 Reference Manual, publication SR-0011.

## SECURE directive

The SECURE directive defines the absolute module (the dataset specified by the ABS directive) to be *secure*. That is, the SECURE directive specifies that this dataset is to be released during job advancement, unless automatic release is specifically overridden with an F$DSD operating system request. (See section 6 for information about how segment datasets can be released.) If you do not use the SECURE directive, SECURE=OFF.

Format:




Parameters:

ON Specifies secure absolute module

OFF Specifies nonsecure absolute module

Continuation on next line is not allowed.

## MEMORY MANAGEMENT GLOBAL DIRECTIVES

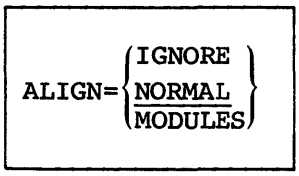
Memory management directives are the following:

- ALIGN
- BCINC
- PADINC
- NORED

### ALIGN directive

The ALIGN directive controls the starting locations of modules and common blocks. SEGLDR sets an align bit for each relocatable module and common block that contains the ALIGN pseudo-op (see the CAL Assembler Version 1 Reference Manual, CRI publication SR-0000) or ALIGN compiler directive (see the FORTRAN (CFT) Reference Manual, CRI publication SR-0009).

Format:




Parameters:

IGNORE Allocates each module and common block to begin at the word following the previous module or common block, ignoring the align bit

NORMAL Allocates each module and common block with the align bit set to an instruction buffer boundary.† If the align bit is not set for a module or common block, that module or common block is allocated at the word following the previous module or common block. ALIGN=NORMAL is assumed if no ALIGN directive is specified.

MODULES Allocates every module to an instruction buffer boundary.† Common blocks are forced to instruction buffer boundaries only if the align bit is set.

## BCINC directive

The BCINC directive specifies the blank common increment value. This value is a decimal count of the number of words by which the size of blank common is to be increased when the program is loaded for execution.

Format:

```
BCINC=nnn
```

Parameter:

*nnn* Size (decimal word count) of blank common increment

By default, the value of the blank common increment is 0.

You cannot continue this directive on a second line.

## PADINC directive

The PADINC directive specifies the pad increment. This value is a decimal count of the number of words of unused space to be made available to the job when the program is loaded for execution. After the program is loaded with its requested extra space, the job is placed in user-managed field length reduction mode (see the MEMORY control statement and the memory management section in the CRAY-OS Version 1 Reference Manual, publication SR-0011) for the duration of the job step.

† Instruction buffer sizes are the following.

| | |
|---|---|
| CRAY-1 | 16 words |
| CRAY X-MP | 32 words |

Format:

```
PADINC=nnn
```

Parameter:

*nnn* Size (decimal word count) of pad increment

By default, the value of the pad increment is 0.

You cannot continue this directive on a second line.

### NORED directive

The NORED directive specifies no field length reduction. Before the program is loaded, the job is placed in user-managed field length reduction mode for the duration of the job step. If you do not use the NORED directive, NORED=OFF.

Format:

```
NORED={ON }
      {OFF}
```

Parameters:

ON Disables memory reduction

OFF Allows memory reduction

## HEAP MEMORY MANAGEMENT GLOBAL DIRECTIVES

The HEAP and STACK directives manage heap memory. (For more information on heap memory management, see the CRAY-OS Version 1 Reference Manual, publication SR-0011.)

### HEAP directive

The HEAP directive allocates memory that the heap manager can dynamically manage. All memory managers share a common heap. The HEAP directive allows their memory use within a job to increase. The heap is physically

located in memory after the segment tree occupying the largest amount of memory unless the LOWHEAP directive is used. If the DYNAMIC directive is specified, dynamic common is located after the heap, and the heap has a fixed size.

Format:

```
HEAP[=init[+inc][>min]]
```

Parameters:

*init* Initial number of decimal words available to the heap manager. The default is an installation parameter.

*inc* Increment size, in decimal words, of a request to the operating system for additional memory if the heap overflows. A value of 0 indicates that heap size is fixed. If you specify the DYNAMIC directive, SEGLDR ignores an increment size other than 0. The default is defined by an installation parameter.

*min* Size, in decimal words, of the smallest block that can be left on the list of available space on the heap. *min* must be at least 2. The default is defined by an installation parameter.

You cannot continue this directive on a second line.

STACK directive

The STACK directive allocates heap memory to a stack for use by re-entrant CFT and CAL programs. The HEAP directive is not needed except to change the default heap values.

Format:

```
STACK[=init[+inc]]
```

Parameters:

*init* Initial size, in decimal words, of a stack. If *init*≤128 or is absent, an installation parameter is used.

*inc* Size, in decimal words, of additional increments to a stack if the stack overflows. Zero implies that overflow is prohibited. An installation parameter defines the default increment value.

You cannot continue this directive on a second line.

LOWHEAP directive

The LOWHEAP directive causes SEGLDR to physically allocate the Heap Memory in memory before the code, rather that after the largest segment. LOWHEAP implies that the heap has a fixed size (cannot be expanded). The initial size of the heap and minimum size of a block in the heap, can still be specified with the HEAP directive.

You cannot continue this directive on a second line

Format:


MISCELLANEOUS GLOBAL DIRECTIVES

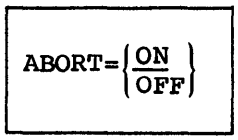
The following global directives apply to either the object module or the binary input modules.

- ABORT
- FORCE
- ORG
- REDEF
- SAVE
- SID
- SYMBOLS

ABORT directive

The ABORT directive controls whether SEGLDR issues a job step abort for certain error conditions. The ABORT directive controls only whether the job step aborts when execution is complete. If loading errors are found, an object module is not created. If you do not use the ABORT directive, ABORT=ON.

Format:




Parameters:

ON Causes SEGLDR to abort if there are errors

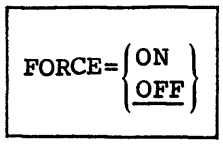
OFF Causes SEGLDR to terminate normally even if there are errors

You cannot continue this directive on a second line.

FORCE directive

SEGLDR gathers the entry points of all modules in all datasets specified with global BIN and LIB directives. It then discards all modules that are never called. The FORCE directive allows subprograms not called by other subprograms to be loaded anyway (force loaded). Only subprograms appearing in segment BIN datasets can be force loaded. If you turn on this option, SEGLDR loads all modules in segment BIN files. If you turn off this option, SEGLDR discards any module appearing in a file specified on a BIN directive, if there are no references to the module (except anything on a XFER directive and BLOCKDATA subprograms). If you do not use the FORCE directive, FORCE=OFF.

Format:




Parameters:

ON Enables segment force-loading

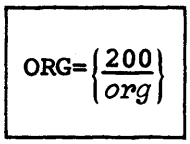
OFF Disables segment force-loading

You cannot continue this directive on a second line.

## ORG directive

The ORG directive sets the load address of the first word of the root segment. This directive is used for debugging purposes; modules being built for use under COS should typically have ORG set to 200. If you do not use the ORG directive, ORG=200.

Format:




Parameter:

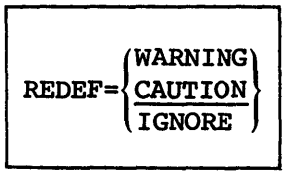
*org* Octal value between 0 and 77777777

You cannot continue this directive on a second line.

## REDEF directive

REDEF controls whether common blocks redefined with different lengths by different modules are treated as loading errors. SEGLDR always takes the longest definition regardless of the REDEF value. If you do not use the REDEF directive, REDEF=CAUTION.

Format:




Parameters:

WARNING Redefinition of common block size produces a warning message.

CAUTION Redefinition of common block size produces a caution message but is otherwise ignored.

IGNORE SEGLDR ignores the redefinition of common block size and does not issue a message.

You cannot continue this directive on a second line.

SAVE directive (global)

The global SAVE directive determines whether the current segment states are written to mass storage before they are overlayed with another segment. The global SAVE directive suppresses or enables saving of all segments.

If you do not use the SAVE directive, SAVE=OFF.

In all other respects it follows the conventions of the local SAVE directive described later in this section.

Format:

SAVE={ON / OFF}

Parameters:

| | |
|---|---|
| ON | Enables segment saving |
| OFF | Suppresses segment saving |

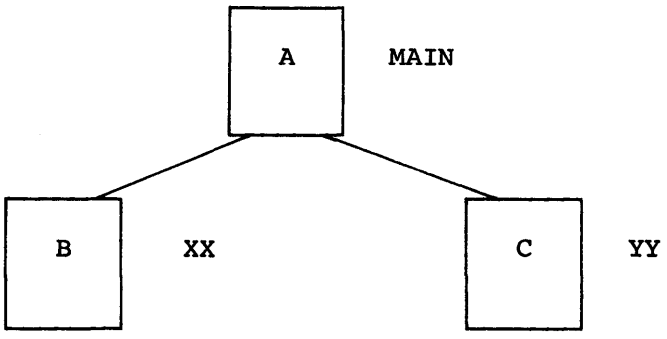
You cannot continue this directive on a second line.

Example:

Consider a program that performs calculations on two large data arrays (X(100000) and Y(100000)). It completes part of the calculations on one array, then on the other, back to the first, and so on.

In this example, arrays X and Y are in subroutines XX and YY respectively. The program is structured as follows:

```
TREE
A(B,C)
ENDTREE
SEGMENT=A
MODULES=MAIN
SEGMENT=B;SAVE=ON
MODULES=XX
SEGMENT=C;SAVE=ON
MODULES=YY
ENDSEG
```



The two arrays can then be overlaid, rather than forced to the root segment (A).

SID directive

The SID directive indicates that this load is for debugging and should include all modules needed for the COS Symbolic Interactive Debugger (SID) to execute. In addition, all symbol table information needed by SID will be written to the dataset $DEBUG. If the SYMBOLS=*dn* directive is used, the symbol table information for SID is written to the dataset *dn*.

Format:

```
SID
```

You cannot continue this directive on a second line.

SYMBOLS directive

The SYMBOLS directive determines whether SEGLDR ignores program symbol table information that might appear in binary input modules or constructs a debug symbol table dataset. If you do not use the SYMBOLS directive, SYMBOLS=ON.

Format:

```
SYMBOLS={ON  }
        {OFF }
        {dn  }
```

Parameters:

| | |
|---|---|
| ON | Writes symbol table information to the $DEBUG dataset |
| OFF | Instructs SEGLDR to ignore all symbol table information. |
| *dn* | Writes symbol table information to the dataset with the name *dn*. ON or OFF cannot be used as dataset names. |

You cannot continue this directive on a second line.

## SEGMENT TREE DEFINITION DIRECTIVES

You use the segment tree definition directives to convey to SEGLDR the shape of the tree that represents the memory layout of your code.

Tree structures can be any width or depth but must be fewer than 1000 segments.

Tree definition directives apply only to segmented codes. The segment tree definition directives are as follows:

- TREE
- ENDTREE

The TREE directive signals the end of the global directive group and the beginning of the segment tree definition group of directives. TREE is followed by the set of directives that specify the tree structure.

The ENDTREE directive terminates this group of directives. Ordering of segment tree definition directives between TREE and ENDTREE is unimportant. The ENDTREE directive signals the end of the tree description and is immediately followed by the segment description directives.

Format:

```
TREE
```

```
segname(segname1,segname2,segname3,...segnamen)
```

```
ENDTREE
```

Parameters:

*segname* Names a segment

$segname_i$ Names all immediate successor segments

If you continue this directive beyond one line, end each line to be continued with a comma.

Example:

Figure 4-1 is the segment tree that corresponds to these directives.

```
TREE
A(B,C)
B(D,E,F)
C(G,H)
G(I,J)
ENDTREE
```

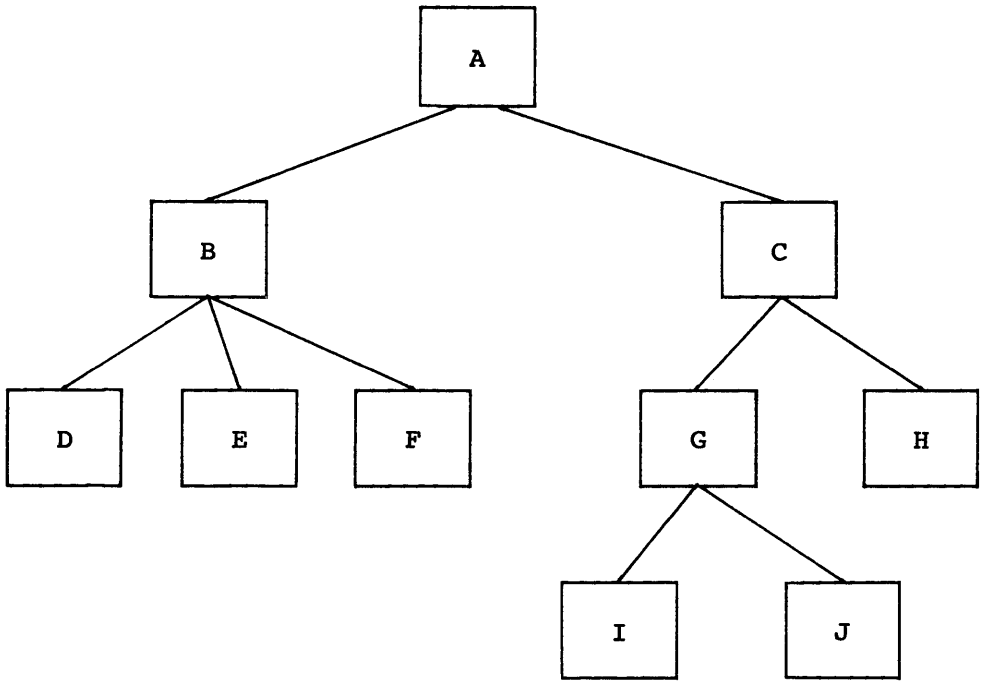



Figure 4-1. Segment tree defined by the preceding set of directives

## SEGMENT DESCRIPTION DIRECTIVES

Segment description directives apply only to segmented codes. Use them to specify the contents of the segments. Assign at least one module per segment.

Assign segment contents (modules) to specific segments using the segment description directives. SEGLDR discards modules that you assign to a segment if there are no calls to them. (To override discarding, use the FORCE directive.)

The segment description directives are the following.

- BIN
- COMMENT
- COMMONS
- DEVICE
- DUP
- ECHO
- ENDSEG
- MODULES
- SAVE
- SEGMENT
- TITLE

---

NOTE

The DUP directive must precede all SEGMENT directives when duplicate entry point names are to be loaded.

---

This subsection does not describe the COMMENT, ECHO, and TITLE directives. See the global directives for their descriptions.

## SEGMENT DESCRIPTION BIN DIRECTIVE

The segment description BIN directive names binary input datasets containing modules to be loaded into a specific segment. SEGLDR loads all modules within the specified BIN datasets into the segment named by the accompanying SEGMENT directive. This directive is the same as global BIN except that modules specified by this directive are fixed (assigned to a segment). Using both BIN and MODULES directives to name the same entry points causes a fatal error.

SEGLDR processes datasets in the order presented. It processes only the first file of each dataset and ignores remaining files without comment.

The effect of multiple BIN directives is cumulative.

Format:

BIN=$bin_1$,$bin_2$,$bin_3$,...,$bin_n$

Parameter:

$bin_i$ Names a relocatable binary dataset or a BUILD library dataset

If you continue this directive beyond one line, end each line to be continued with a comma.

Example:

In the following example, all modules in datasets SEG1A, SEG1B, and SEG1C are loaded into segment SEG1.

```
SEGMENT=SEG1
BIN=SEG1A,SEG1B
BIN=SEG1C
ENDSEG
```

COMMONS DIRECTIVE (LOCAL)

The COMMONS directive names common blocks to be loaded into the segment named by the accompanying SEGMENT directive. Common block specification is optional, except if common blocks are to be duplicated or loaded in a specific order.

This directive overrides the common block floating algorithm (see Appendix B). You may use the PRESET directive to override presetting of common blocks to 0.

Common blocks loaded into two or more segments are considered unique. They occupy different memory locations and the program can reference their contents unambiguously. You may not include the dynamic common block in a COMMONS directive, since it is not assigned to a segment. See section 5 for more detail on common blocks.

Format:

COMMONS=$com_1$,$com_2$,...$com_n$

Parameter:

*com$_i$* Names common blocks to be loaded in the segment named by the accompanying SEGMENT directive

Common blocks are loaded in the order they are specified.

Common blocks are also loaded according to type; the default loading order is as follows: labeled common, code block, and finally blank common. This order can be overridden by the ORDER directive, as shown in the following example.

If you continue this directive beyond one line, end each line to be continued with a comma.

Example:

```
ORDER=BCL; COMMONS=A,B,//,C
```

The resulting order would be blank common (//), the code block, and finally labeled common blocks A,B,C.

## DEVICE DIRECTIVE

SEGLDR assigns one segment to one dataset at execution time. Use the DEVICE directive to name the logical device on which the dataset begins. If you do not use the DEVICE directive, COS chooses a logical device. Consult Cray site operations for possible logical device names.

Execution of a segmented program produces temporary datasets called *segment datasets*, each of which contains a single segment. See section 6 for more information about segment datasets and code execution.

Format:

```
DEVICE=device
```

Parameter:

*device* Device name; 1- to 8-character string (for example, DEVICE=DD-19-21).

See your CRI site analyst for valid device names.

## DUP DIRECTIVE

Use the DUP directive if you require duplicate entry point names. You use it to load several copies of the same module or more than one module with the same entry point name. Duplicate entry point handling is discussed in detail in Appendix A.

Format:

DUP=*entname*(*seg*$_1$,*seg*$_2$,...*seg*$_n$)

Parameters:

*entname* Name of an entry point to be loaded in more than one segment

*seg*$_i$ Names of the segments where *entname* is to be loaded

Example 1:

The following is an example of a segment description directive grouping that includes the DUP directive.

```
DUP=SUBX(SEG1,SEG2)
SEGMENT=SEG1
MODULES=SUBY
COMMONS=COMBLK1
ENDSEG
SEGMENT=SEG2
MODULES=SUBZ
COMMONS=COMBLK1
ENDSEG
```

The following figure is an example of the above tree with duplicate entry point assignment. It assumes that the module name and entry point name are the same. Entry point SUBX is duplicated in segments SEG1 and SEG2.

Note that if SUBY is to call SUBX in segment SEG1, SUBY must be assigned to segment SEG1. If SUBY is to call SUBX in segment SEG2, SUBY must be assigned to segment SEG2. If SUBY were to go into the root, the call would be ambiguous.

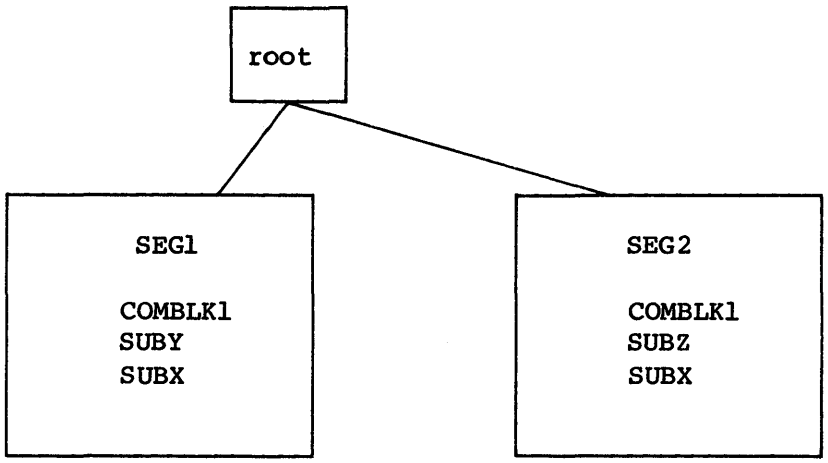



Figure 4-2. Tree with duplicate entry points

Also note that common blocks are duplicated in different segments by the use of the COMMONS directive. If you do not want exactly one copy of a common block you must use the COMMONS directive to place the common blocks in the segments desired.

MODULES DIRECTIVE (LOCAL)

A MODULES directive allows you to specify modules to be assigned to the segment named by the SEGMENT directive.

Format:

MODULES =*modname*$_1$,*modname*$_2$,...*modname*$_n$

Parameter:

*modname*$_i$ Name of modules to be loaded

You may specify the parameter *modname*$_i$ as either *modname* or *modname*:*dsname*. Use the second form to name a module to be loaded when also specifying the binary dataset or library from which to obtain the module.

Example:

```
MODULES=SUBA, SUBB:LIB1,SUBC,SUBD:DTASET1
```

SEGLDR obtains modules SUBA and SUBC according to the normal loading order. It obtains SUBB from library LIB1 and SUBD from binary input dataset DTASET1.

## SAVE DIRECTIVE (LOCAL)

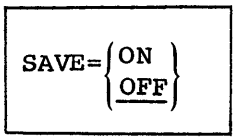
The local SAVE directive specifies whether the current segment state is written to mass storage before SEGLDR overlays it with another segment. This directive overrides the effect of the global SAVE directive on individual segments.

If you do not use the SAVE directive, SAVE=OFF. If the SAVE directive is OFF when a segment is loaded into the same memory area as the current segment, the updated values in the current segment are lost.

If the SAVE directive is ON, however, SEGLDR writes the updated image of the overlayed segment to mass storage before the new segment is loaded.

Subsequent execution of a saved segment starts from it's image. This enables you to overlay large data arrays whose updated values you require in subsequent executions.

Format:




Parameters:

| | |
|---|---|
| ON | Enables segment saving |
| OFF | Suppresses segment saving |

You cannot continue this directive on a second line.

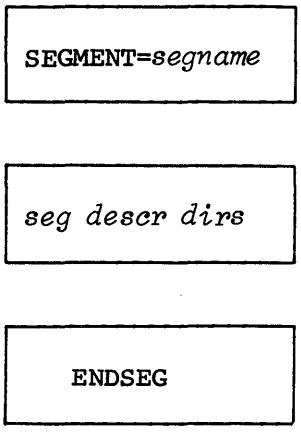
For an example of the use of this directive, see global SAVE in this section.

SEGMENT DIRECTIVE

The SEGMENT directive names the segment being described by the segment description directives.

SEGMENT is always the first of the segment description directives, except when using the DUP directive, which must precede SEGMENT. ENDSEG terminates the segment description. Between SEGMENT and ENDSEG are any of the rest of the segment description directives, in any order. Note that a MODULES or BIN segment description directive must always be associated with a SEGMENT directive, since you must assign at least one module to each segment.

Format:




Parameter:

*segname* Segment name; 1-8 characters.

Example:

```
SEGMENT=SAM
SAVE=OFF
MODULES=A,B,C; COMMONS=//,SAMCOM
ENDSEG
```

# COMMON BLOCK USE AND ASSIGNMENT 5

Common blocks can be assigned to segments either by you or by SEGLDR.

## USER-ASSIGNED COMMON BLOCKS

You specify the segment in which to load a common block with the COMMONS segment description directive. All common blocks named in a single COMMONS directive are assigned to the same segment. This segment is named by the SEGMENT directive. The common blocks named are loaded in the indicated order.

You can cause more than one common block of the same name to be allocated. The blocks must be in different segments, and references to them must be unique (each requires its own COMMONS segment description directive).† You cannot assign two copies of a common block to two segments on a common branch. Also, if copies of a common block are specified in two segments on different branches, they cannot have a common caller. For example, if common block /ABC/ were included in segments B and C in the segment tree in figure 5-1, a reference to /ABC/ from a module in segment A would be ambiguous. (The table in Appendix A further illustrates this restriction.)

In figure 5-1, assume a copy of /ABC/ has been included in both segments B and C. References from segments C, D, and E would be relocated to the /ABC/ common block in segment C. References to /ABC/ from segment B would be relocated to the /ABC/ common block in segment B.

## SEGLDR-ASSIGNED COMMON BLOCKS

All common blocks without a user-assigned location are considered movable blocks. They are assigned to segments according to the following algorithm: a common block is assigned to a segment that is the nearest

† All subprograms referencing the duplicated common blocks must be assigned by you rather than by SEGLDR.

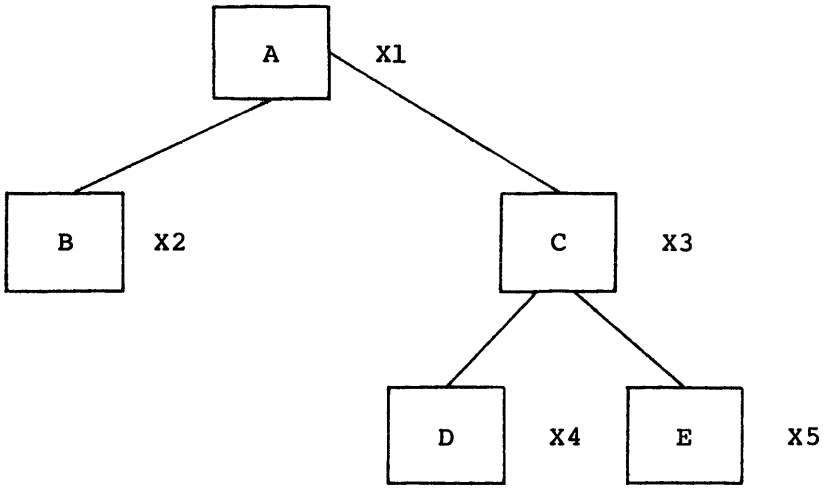



Figure 5-1. Segment tree

common predecessor of all the segments that reference that common block. For example, if, in the preceding figure X2 and X4 reference /ABC/, SEGLDR assigns it to segment A. If X4 and X5 reference /ABC/, SEGLDR assigns it to segment C.

Movable block assignment is further discussed in Appendix B.

## COMMON BLOCK SIZES

You may redefine the length of any common block at any time. SEGLDR uses the longest common block found from all modules encountered for each named or blank common block loaded. SEGLDR generates a message if a conflicting common block size is encountered. You may control the severity of this message with the REDEF directive. REDEF also allows supression of the message (see section 4).

## DUPLICATE COMMON BLOCKS

Common blocks loaded into two or more segments are considered unique, since they occupy different memory locations. Modules that reference a duplicated common block must be assigned to ensure that the program contains no ambiguous references to common block data. (See the COMMONS segment description directive in section 4.)

## DATA LOAD RESTRICTIONS

Data loads (see the glossary) from segments other than the segment where the common block resides are not processed. SEGLDR issues warning messages for data loads from other segments and skips the data. Dynamic common and blank common cannot be data loaded.

**********

CAUTION

SEGLDR does not permit you to do everything with common blocks that you can do within CAL. Relocation of values within common blocks is not permitted. SEGLDR treats such cases as loading errors. Entry points, however, can be declared in common blocks.

**********

## BLOCK DATA ROUTINES

If a module in a BIN dataset is a block data routine, SEGLDR always loads it.

If block data in LIB routines is to be loaded, it must be referenced by a previously loaded program or in a MODULES directive.

If you have a subroutine (not block data) that is never called but contains data loads, use the FORCE directive (see section 4), or the EXTERNAL statement (see the FORTRAN (CFT) Reference Manual, CRI publication SR-0009), to be sure it is loaded.

## REFERENCING DATA IN COMMON BLOCKS

Data in a common block can be referenced by any module that is in either the same or a predecessor segment. For data in a common block to be referenced by a module in a successor segment, however, all of the following conditions must be true to avoid getting an error. Note that these conditions cannot be verified at load time.

1. The COMMONS directive must be used in the successor segment to fix the common block with the data being referenced.

2. A call must be made to the successor segment before the data reference is made.

3. No subsequent calls can be made to other segments that would cause the original successor segment to be overlayed.

# CODE EXECUTION 6

A segmented program is called into execution by a COS control statement using the local dataset name of the absolute binary dataset as the verb. Additional control statement parameters can also be provided, the same as for nonsegmented programs.

Execution of a segmented program produces temporary datasets called *segment datasets*. These datasets are created by the resident routine ($SEGRES) as part of its initialization. Segment datasets are named $SEG001, $SEG002,...,$SEG*nnn*, where *nnn* is the number of branch segments.

Each segment dataset consists of a single record containing a segment. Segment datasets are read and written with COS-buffered I/O requests made by $SEGRES. You can name the physical device to which a segment is to be assigned with the segment description DEVICE directive.

At the beginning of execution, COS transfers control to the SEGLDR-resident routine, $SEGRES, at entry point $SEGRES. $SEGRES is a system routine loaded with the object module. $SEGRES is designed to efficiently read segments to memory for execution and write segments to mass storage to allow saving current segment states.

$SEGRES contains only two entry points, $SEGRES and $SEGCALL, and a single common block--/$SEGRES/.

$SEGRES accepts control from the COS Control Statement Processor (CSP) when execution begins and is responsible for some initialization functions. $SEGRES begins by initializing the segment datasets: Each nonroot segment is copied from the absolute binary dataset to its own segment dataset. Using one dataset per segment has two advantages:

- Individual segments can be most efficiently read and written at run time because the amount of I/O is at a minimum.
- You can control the logical device assignment at run time, assigning different segments to different devices. See the ASSIGN control statement in the CRAY-OS Version 1 Reference Manual, publication SR-0011. (Note that you do not use this method of assigning a segment to a device if you are using the DEVICE directive.)

Following the copy operations, control transfers to the entry point you specify.

$SEGCALL intercepts subroutine calls that might require segments to be loaded to memory. $SEGCALL also saves memory-resident segments, so they are not overwritten, if SAVE=ON for those segments.

$SEGRES is small in size, approximately $12000_8$ words, including two $4000_8$ I/O buffers. Other special features of $SEGRES described in this section are:

- Minimal overhead for subroutine calls
- I/O streams for loading and unloading segments (user-specified devices for individual segments)
- Two modes for segmented program memory management

With termination of the segmented program, the segment datasets can be released all at once by invoking the SEGRLS utility. SEGRLS is executed by the following COS control statement:

```
SEGRLS.
```

## SUBROUTINE CALL OVERHEAD

With SEGLDR, four levels of subroutine call overhead are possible. In order of increasing run time, they are:

1. No $SEGRES intervention: Callers and callees are in the same segment. In addition, no intervention is required for subroutine calls with the callee in a predecessor segment (the callee is memory resident at the time of the call).
2. The callee is in a successor segment that is memory resident. $SEGRES immediately notes that the callee is in memory and passes control to the callee.
3. The callee is in a successor segment that is not in memory; one or more successor segments are read to memory.
4. The callee is in a successor segment that is not in memory and SAVE=ON is specified; one or more currently resident segments are written to disk so they are not overwritten when the called segment is written to memory.

## I/O PERFORMANCE

Individual segments are copied from the object module dataset to individual datasets ($SEG001, $SEG002,...) as part of $SEGRES initialization, allowing you to assign I/O streams to individual segments.

$SEGRES initiates two read or two write streams, allowing concurrent loading or saving of two segments. Only a single-buffered I/O request is required to initiate transfer of segments to or from mass storage once a stream assignment is made.

Separate datasets for each code segment allow you to assign specific devices to individual segments, optimizing the I/O performance of $SEGRES.

## MEMORY MANAGEMENT

$SEGRES provides for two memory management modes: static and dynamic.

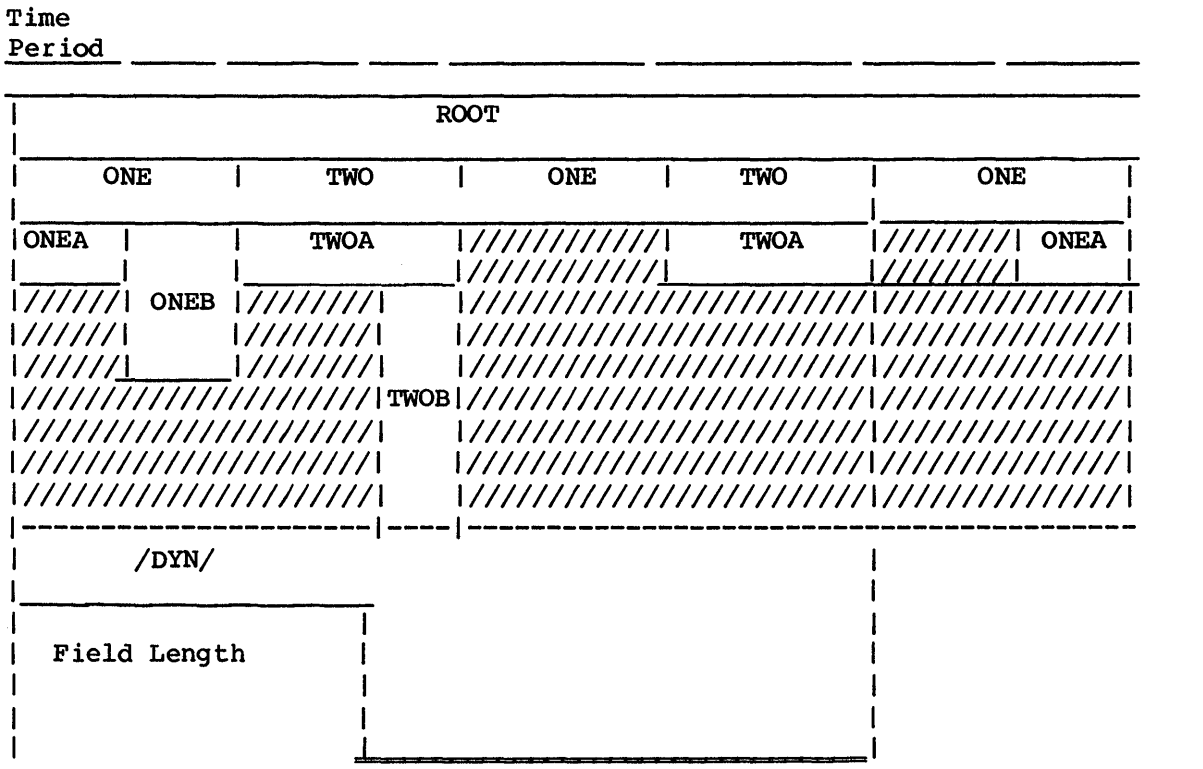
### STATIC MEMORY MANAGEMENT

Static memory management is in use when you declare a dynamic common block (see the DYNAMIC directive in section 4). $SEGRES sets HLM (high limit memory) to the end of the dynamic common block. $SEGRES never resets HLM following initialization. However, HLM may be altered by the execution of your code.

$SEGRES neither requests additional memory nor releases excess memory. Memory field length is completely under the control of your program. If the field length is reduced by COS calls in your program, the program must ensure that field length is large enough to read segments for execution when necessary.

The following example illustrates static memory management (you declare a dynamic common block):

```
DYNAMIC=DYN
TREE
ROOT(ONE,TWO)
ONE(ONEA,ONEB)
TWO(TWOA)
TWOA(TWOB)
ENDTREE
```

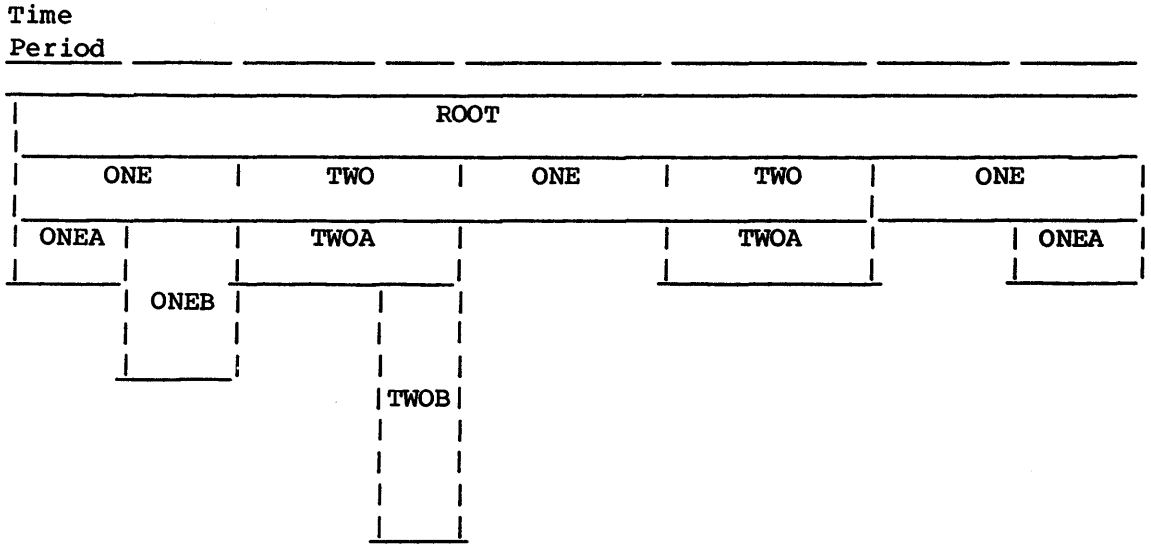



## DYNAMIC MEMORY MANAGEMENT

Dynamic memory management is used when no dynamic common block is present. HLM is set by SEGLDR to the end of the terminal segment of the segment tree branch in memory. Segment loading and unloading activity causes $SEGCALL to request memory from or relinquish memory to COS as required.

The following example illustrates dynamic memory management ($SEGRES manages field length):

```
TREE
ROOT(ONE,TWO)
ONE(ONEA,ONEB)
TWO(TWOA)
TWOA(TWOB)
ENDTREE
```



# APPENDIX SECTION

# DUPLICATE ENTRY POINT HANDLING

A

You can cause multiple copies of the same entry point or more than one entry point of the same name in different modules to be loaded in different segments.

SEGLDR must know where to place all duplicate entry point names before encountering modules that call them, so that it can link to the proper entry point. You then, rather than SEGLDR, must assign all duplicated entry points and their callers, and all modules that refer to duplicated common blocks.

Avoid assigning two copies of an entry point or common block to two segments on a common branch. If you specify copies of an entry point or common block in two segments on different branches, ensure that they do not have a common caller. Table A-1 illustrates the restrictions.

Table A-1. Segment assignments in tree form

| Duplicated module X in segments in figure A-1 | Module X Called From | Comments |
|---|---|---|
| B,D | B,C | Calls from B are linked to the copy in B. Calls from C are linked to the copy in D. |
| C,E | Anywhere | Illegal. All calls would be ambiguous. |
| B,C | B,C,D,E | Calls from B are linked to the copy in B. All others are linked to the copy in C. |
| B,C | A | Illegal. Reference is ambiguous. |
| B,B | Anywhere | Illegal. Cannot have two copies in the same segment. |

You must include the entry point and all secondary entry points of any duplicated module in a DUP directive (see section 4 for more information on the DUP directive).

Consider the following tree:

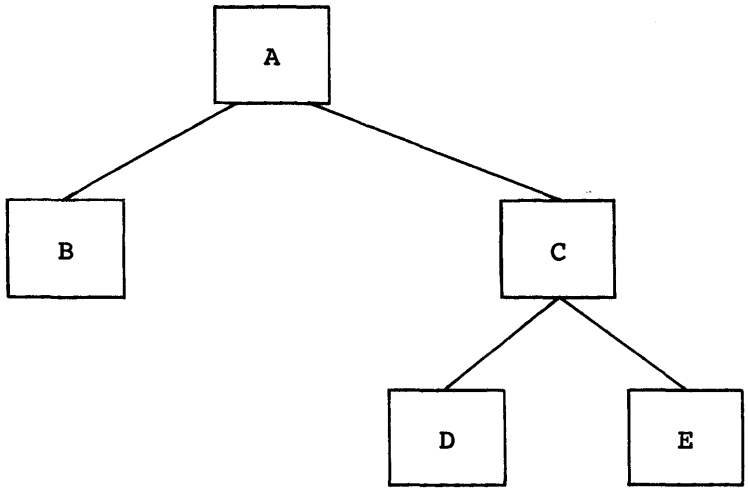



Figure A-1. Sample tree

In figure A-1, suppose that entry point X in segment B is in dataset BIN1 and that another X in segment E is in BIN2. Assume that W calls the X in segment B and Y calls the X in segment E. Further, assume that the X in segment B has a secondary entry point, X1. The directives required to obtain this description are the following.

```
TREE
  A(B,C)
  C(D,E)
ENDTREE
DUP=X(B,E)
DUP=X1(B)
SEGMENT=A
  BIN=BIN3
ENDSEG
SEGMENT=B
  MODULES=X:BIN1,W
ENDSEG
SEGMENT C
  MODULES=Y
ENDSEG
SEGMENT=D
  MODULES=XYZ
ENDSEG
SEGMENT=E
  MODULES=X:BIN2
ENDSEG
```

# MOVABLE BLOCK ASSIGNMENT BY SEGLDR

B

With SEGLDR, a movable block is a module or common block not assigned to a specific segment in a segment description directive. SEGLDR treats as movable all blocks whose locations (segments) you do not specify, and assigns movable modules or common blocks to segments based on all subprograms that reference them. SEGLDR floats movable blocks to the highest-level segment preceding all callers.

In figure B-1, for example, if movable module X is called by X4 and X5, it is assigned to segment C. If movable module X is called by X2 and X4, it is assigned to segment A. If X is called by X1, it is assigned to segment A. If X has no callers, it is assigned to segment A.

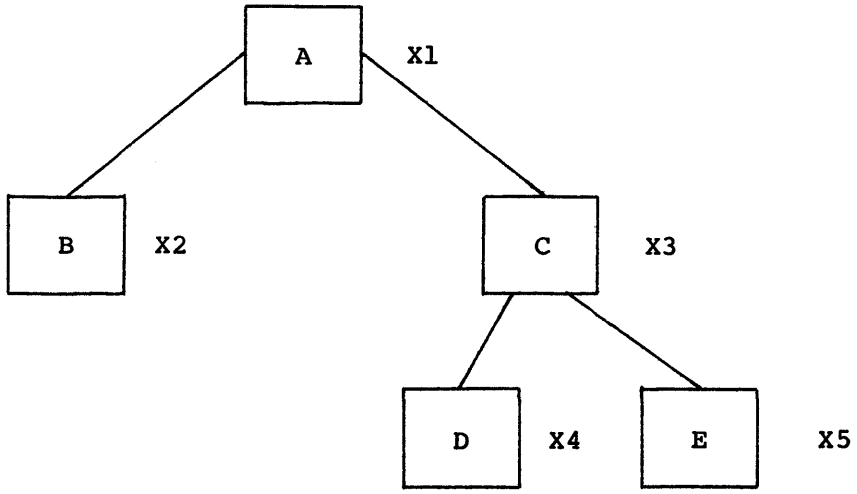



Figure B-1. Sample segment tree

A call to a movable module cannot cause a segment to be loaded. (If it could, a large number of loads could occur at execution time, incurring a large I/O charge.)

A movable common block is assigned to the nearest common predecessor of all segmemts containing references to it. Common blocks without references to them are not loaded. Such a common block might be one appearing in a library routine that was discarded during call tree trimming. (For more information on common blocks, see section 5.)

# REDUNDANT ENTRY POINTS C

To satisfy external references from more than one binary file or library, two or more modules may be loaded, resulting in redundant entry points. For example, assume there are external references to entry points A and B. Assume there is a module A in datasets BIN1 and BIN2 as follows:

A in BIN1 has no secondary entry points.

A in BIN2 has secondary entry point B.

Further assume that BIN=BIN1,BIN2 appeared as a global directive. SEGLDR loads module A from BIN1 because of the external reference to A and because BIN1 is considered first. Module A in BIN2 is also required because of the external reference to B (a secondary entry in A).

In this and other similar circumstances, SEGLDR loads both versions of module A. The second occurrence of the entry point A (in BIN2) is considered inactive.

An informative diagnostic in the load map reports all inactive entry points. If the FULL option in the load map is specified (see the MAP directive in section 4), the redundant entry points appear in the entry point cross-reference map with the notation "inactive". See Appendix G for load map information.

# EXTENDED BLOCK RELOCATION D

Extended block relocation is the adjustment of a 1- to 64-bit field within a module. The field need not begin on a parcel or word boundary. SEGLDR checks to ensure that the relocation quantity to be added will not overflow the defined field. As an example, consider the following CAL code sequences.

Example 1: Relocation of the *jkm* field of an 020 instruction

```
          IDENT          ALEX
                    :
                :
HERE      =         *
                :
                :
          A0        HERE
                :
                :
          END
```

SEGLDR processes the A0 HERE instruction's *jkm* field as an extended block relocation since HERE is a label local to module ALEX. If the relocation quantity to be added (the base address of module ALEX as assigned by SEGLDR) results in a value exceeding the reserved 22-bit field, SEGLDR issues an error.

Example 2: Relocation of a negative quantity within a 10-bit field

```
          IDENT          SALLIE
                :
                :
HERE      BSS       1
                :
                :
          VWD       D'8/0,D'10/HERE-O'1000,D'46/0
                :
                :
          END
```

Suppose that CAL assigns HERE a relocatable value of O'700. Then the expression HERE-O'1000 has the relocatable value of -O'100. If the base of module SALLIE is assigned location O'10000, the adjusted value of the expression is O'7000, which exceeds 10 bits, generating a SEGLDR error. If SALLIE is assigned to location O'2077, however, the expression value will be O'1777, which occupies 10 bits.

---

NOTE

CAL truncates relocatable values that overflow their assigned fields at assembly time.

---

# TYPICAL LOADS AND TREE STRUCTURES

E

This appendix presents examples of some typical loads and segment tree structures with their corresponding sets of directives.

Example 1:

The FORTRAN program in this example is compiled, loaded, and executed beginning at entry point START. The program produces a full load map. Its source is in the dataset SOURCE. The object module is nonsegmented.

```
CFT,I=SOURCE,B=BINS.
SEGLDR.
TEST.
/EOF
BIN=BINS
ABS=TEST
MAP=FULL
XFER=START
```

Example 2:

This example is based on the tree structure in figure E-1. Given this tree structure, assume that all entry points in binary dataset BIN1 are to be loaded in segment C and all entry points in BIN2 in segment E. All other entry points are to be obtained from global binary datasets BIN3 and BIN4 and the default libraries. Entry points Y, W, and Z are in segments A, B, and D, respectively. Also assume that segments B and C contain large data arrays whose updated values are needed each time they are executed. Assume that version 1 of entry point X (in BIN3) is needed in segment D and version 2 (in BIN4) in segment F. All calls to entry points Y1, Y2, and Y3 are to be linked to entry point Y.

The control statement and directives included are:

```
SEGLDR,I=INPTS.
```

INPTS contains the following directives:

```
BIN=BIN3, BIN4; EQUIV=Y (Y1,Y2,Y3)
TREE
A(B,C)
C(D,E,F)
```

```
ENDTREE
DUP=X(D,F)
SEGMENT=A
MODULES=Y
ENDSEG
SEGMENT=B;SAVE=ON
MODULES=W
ENDSEG
SEGMENT=C;SAVE=ON
BIN=BIN1
ENDSEG
SEGMENT=D
MODULES=Z,X:BIN3
ENDSEG
SEGMENT=E
BIN=BIN2
ENDSEG
SEGMENT=F
MODULES=X:BIN4
ENDSEG
```

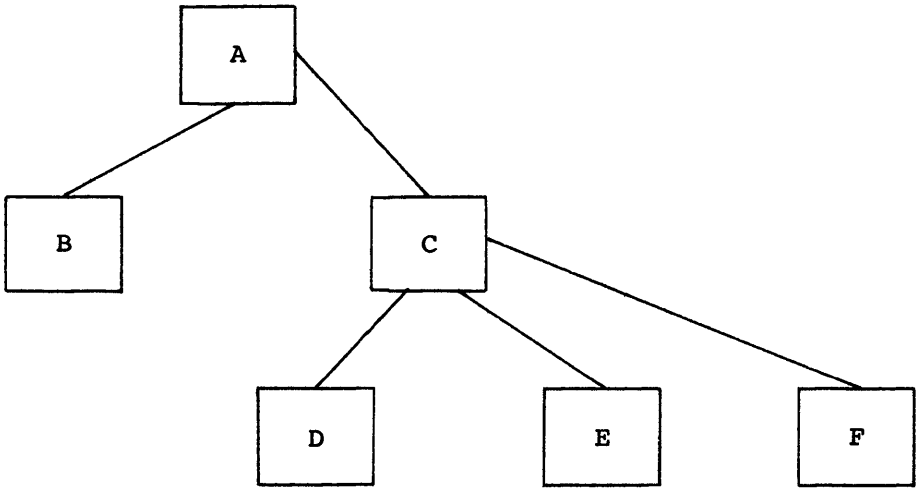



Figure E-1. Sample tree structure

Example 3:

This example is based on the tree structure in figure E-2. Given this tree structure, the tree definition directives are:

```
TREE
B(E,F)
H(I,J,K)
A(B,C,D)
F(G,H)
ENDTREE
```

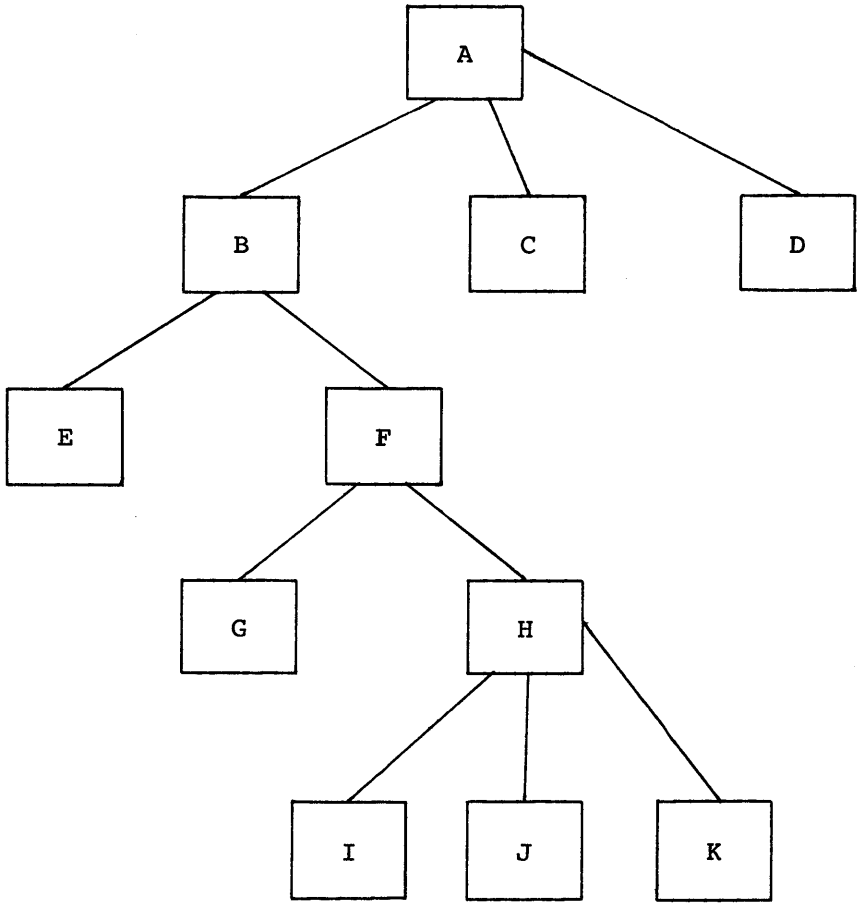



Figure E-2. Sample tree structure

Example 4:

Given the tree structure illustrated in example 3 (figure E-2), assume that a dynamic common block /DYN/ is used and expanded at execution time. All entry points are obtained from a global binary dataset MYBIN, and BLIB and BASELIB. The load order is changed to: blank, labeled, and code. Common block /AA/ is to be assigned to segment J. A full load map on dataset MAP is desired.

The control statement and directives required are:

```
SEGLDR,I=INS,L=MAP.
```

INS contains the following directives:

```
BIN=MYBIN;LIB=BLIB,BASELIB;MAP=FULL
```

```
DYNAMIC=DYN;ORDER=BLC
TREE
A(B,C,D)
B(E,F)
F(G,H)
H(I,J,K)
ENDTREE
SEGMENT=A
MODULES=MAIN
ENDSEG
SEGMENT=B
MODULES=SUBB
ENDSEG
SEGMENT=C
MODULES=SUBC
ENDSEG
SEGMENT=D
MODULES=SUBD
ENDSEG
SEGMENT=E
MODULES=SUBE
ENDSEG
SEGMENT=F
MODULES=SUBF
ENDSEG
SEGMENT=G
MODULES=SUBG
ENDSEG
SEGMENT=H
MODULES=SUBH
ENDSEG
SEGMENT=I
MODULES=SUBI
ENDSEG
SEGMENT=J
COMMONS=AA;MODULES=SUBJ
ENDSEG
SEGMENT=K
MODULES=SUBK
ENDSEG
```

Example 5:

The tree definition directives for the following tree structure (figure E-3) are:

```
TREE
A(B,C,D,E,F,G,H)
ENDTREE
```

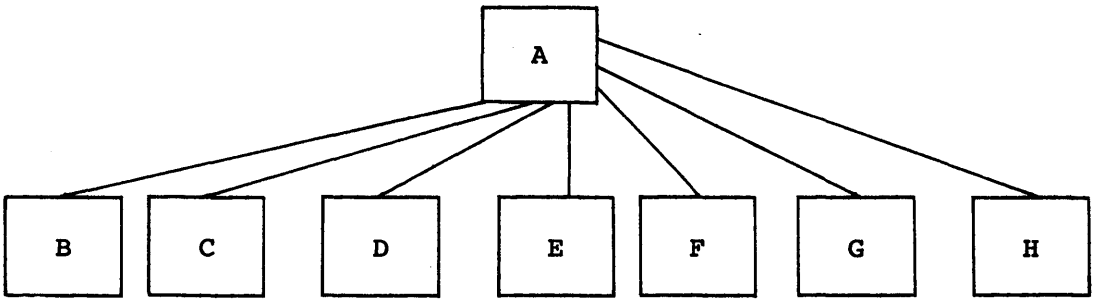



Figure E-3. Sample tree structure

# MESSAGES F

Two types of messages are associated with SEGLDR: the logfile messages and the listing messages.

## SEGLDR LOGFILE MESSAGES

Following is a list of all SEGLDR logfile messages sorted by message ID. SEGLDR messages are prefixed by SG followed by a 3-digit number.

SG000 - SEGLDR VERSION x.nn - *mm/dd/yy*
This logfile message identifies the copy of SEGLDR being used by version number and generation date.

SG010 - ILLEGAL INPUT DATASET NAME- *dn*
SG011 - ILLEGAL LIST DATASET NAME- *dn*
The two messages above flag illegal input and list dataset names. The SEGLDR control statement I or L keyword is equated to an inappropriate dataset name.

SG012 - INPUT DATASET NOT LOCAL- *dn*
This message indicates that the dataset containing SEGLDR directives is nonexistent. The cause is probably a misspelled value for the I control statement parameter or failure to access the input dataset.

SG013 - ILLEGAL DW PARAMETER VALUE - *value*
The width of input directives must be specified as a number greater than 0 and less than 81.

SG015 - GLOBAL DIRECTIVE ERRORS
The SEGLDR global directives contain one or more fatal errors. Examine the listing dataset for specific fatal error diagnostics.

SG017 - SEGMENT TREE STRUCTURE ERROR
The segment tree is faulty. Examine the listing dataset for specific fatal error diagnostics.

SG019 - SEGMENT DESCRIPTION ERRORS
One or more segments are improperly described by a SEGMENT/ENDSEG directive group. Examine the listing dataset for specific fatal error diagnostics.

SG020 - LOAD ERRORS, DN=*dn*

SEGLDR detected one or more loading errors while processing the indicated load dataset. Examine the listing dataset for specific fatal error diagnostics.

SG021 - SEGLDR RESIDENT ROUTINE MISSING

The routine that handles intersegment subroutine calls was not located. Normally this subroutine appears in one of the default libraries. Possible causes of this error are the following.

- The NODEFLIB directive was used and a SEGLDR resident routine is not provided by the user.
- The resident routine is not provided in a system-supplied library (see the Cray Research site analyst).

SG024 - NO TRANSFER ENTRY POINT

The entry point named in the XFER directive is not located in any binary input dataset, or no XFER directive was issued and no primary entry point exists. Be sure to specify the binary input dataset containing the desired transfer entry point.

SG025 - MODULE ASSIGNMENT ERRORS

One or more subroutine linkages between user-fixed modules are improper. Subroutine calls must not call across segment tree branches. A subroutine can call routines in the same segment, predecessor segments, or successor segments. Examine the listing dataset for specific fatal error diagnostics.

SG026 - TRANSFER ENTRY NOT IN ROOT SEGMENT

The entry point specified by the XFER directive is not located in the root segment. Either change the transfer entry point to an entry point in the root segment or include the binary input dataset containing the specified entry point in the root segment.

SG027 - DYNAMIC COMMON BLOCK NOT LOADED

The common block named by the DYNAMIC directive was not found or was discarded by the call tree trimming process.

SG028 - POSSIBLE CALLING SEQUENCE MISMATCH, DN= *dn*

Some included routines may have been compiled with different calling sequence versions. Make sure that all routines are compiled with the same calling sequence version.

SG163 - NULL INPUT FILE

The input dataset from which directives are read contains no SEGLDR directives. Check the spelling of the input dataset name and the contents of the specified dataset.

SG200 - *nnn* UNSATISFIED EXTERNALS
The program that was loaded contains one or more unsatisfied externals. A complete report of unsatisfied external references is contained in the listing generated by SEGLDR for the load.

SG300 - AVAILABLE MEMORY EXHAUSTED
SEGLDR is unable to load the program in the available memory field. Increase the maximum memory field size (the COS JOB control statement MFL parameter).

The following messages (SG801-SG804) are issued by the resident routine $SEGRES at run time.

SG801 - SDT LOOKUP FAILURE
SG802 - SLT LOOKUP FAILURE
Both messages indicate an internal failure of SEGLDR routine $SEGRES. Give the failing job to a Cray Research analyst.

SG803 - /$SEGRES/ DESTROYED
The user's program wrote over the $SEGRES common block. Use the COS-supplied trace-back information or dump information to isolate the code that caused the problem.

SG804 - BUFFER I/O ERROR, DSN = *dn*
A disk malfunction occurred as segments were read to or written from memory during program execution (*dn* identifies the dataset.)

The following logfile messages are issued in response to faulty binary input datasets (both LIB and BIN). *dn* is a dataset name, *n* is an octal number corresponding to the dataset word address where the error was detected, *ii* is a 2-digit octal number between 00 and 17, and *mod* is the name of the bad module if known. One possible cause of these conditions is nonbinary input (a dataset containing textual data).

SG900 - UNEXPECTED READ STATUS, DATASET *dn* POS=*n*
SG901 - FIRST TABLE NOT A PDT, DATASET *dn* POS=*n*
SG902 - BAD READ STATUS OR PDT WC, DATASET *dn* POS=*n*
SG903 - UNEXPECTED READ STATUS, DATASET *dn* MODULE *mod*
SG904 - UNEXPECTED TBL=*ii* DATASET *dn* MODULE *mod*
SG905 - UNKNOWN TBL=*ii* DATASET *dn* MODULE *mod*

The following messages result from a faulty binary module. The cause is an internal error in the assembler or compiler that created the module. SEGLDR does not try to recover from these errors; abort is immediate. Refer the binary input to a Cray Research site analyst.

SG906 - BAD BI FOR MODULE *mod* IN DATASET *dn*
SG907 - PWT REFERENCES IRB, MODULE *mod* DATASET *dn*
SG908 - BAD BI IN DUP. TBL, MODULE *mod*
SG909 - DPT REFERENCES IRB, MODULE *mod*
SG910 - BAD BI IN PWT, MODULE *mod*

SG911 - PWT REFERENCES IRB, MODULE *mod*
SG912 - BAD XI, MODULE *mod*
SG913 - BAD XI IN XRT, MODULE *mod*
SG914 - XRT REFERENCES IRB, MODULE *mod*
SG915 - BAD BI0 IN BRT, MODULE *mod*
SG916 - BRT BI0 REFERENCES IRB, MODULE *mod*
SG917 - BAD BI IN BRT, MODULE *mod*
SG918 - BAD BI IN XBRT, MODULE *mod*

SG919 - ERROR RE-READING ROOT SEGMENT
An error was encountered while reading from the output dataset during creation of the executable dataset.

## LISTING MESSAGES

SEGLDR produces five types of load-time messages that you may specify to be printed on the listing file. From least severe to most severe, they are the following.

| | |
|---|---|
| COMMENT | Prints an informational message that has no effect on the execution of the object module |
| NOTE | Prints a message that indicates that SEGLDR may have been misused or used inefficiently. This error has no effect on the execution of the object module. |
| CAUTION | Prints a message that indicates a possible error has been detected but it is not severe enough to prohibit execution. The object module is written and will execute as desired |
| WARNING | Prints a message that indicates an error that probably invalidates the object module. The output is not written, but processing continues so that additional messages may be printed. |
| ERROR | A fatal error has been detected and processing cannot continue. No object module is written. |

These messages are listed in alphabetical order without regard to severity.

*WARNING* '=' EXPECTED AFTER- *symbol*
The keyword *symbol* is not followed by =. Use the correct directive syntax.

*WARNING* '(' EXPECTED AFTER- *symbol*
Missing or misplaced open parenthesis. Lists must be enclosed in parentheses. Put an open parenthesis in the correct place.

*WARNING* ')' OR ',' EXPECTED AFTER- *symbol*
While processing a list of names, an end of line was encountered before a comma or right parenthesis. List names must be separated by a comma and enclosed by parentheses. If a line continuation is desired, put a comma after the last name on the line.

*WARNING* ',' or ':' EXPECTED AFTER - *modname*
While processing the MODULES directive an unknown delimitor was found after the module named. Use the desired terminator.

*CAUTION* ABSOLUTE MODULE - *modname* IN LOAD DATASET- *dn*; MODULE SKIPPED
An absolute binary module was encountered when a relocatable module was expected. Remove the absolute binary module or do not specify *dn*.

*CAUTION* ADDRESS RELOCATION IN COMMON BLOCK *cbname* BY MODULE *modname*
The common block *cbname* contains code or other data which is preset to a relocatable address. If code is present in a CAL common block, it should be removed and placed in a code module. If data is preset to a relocatable value, the data should be preset with a nonrelocatable value or initialized to the relocatable value desired at execution time.

*WARNING* AMBIGUOUS REFERENCE TO DUPLICATED COMMON BLOCK- *cbname* FROM MODULE- *modname*
Two or more links exist between *cbname* and *modname*. Redesign the segment structure and module assignments to resolve ambiguity or remove a copy of *cbname*.

*WARNING* AMBIGUOUS REFERENCE TO DUPLICATED COMMON BLOCK- *cbname* FROM MODULE- *modname* IN SEGMENT- *segname*
Two or more links exist between *cbname* and *modname*. Redesign the segment structure and module assignments to resolve ambiguity or remove a copy of *cbname*.

*WARNING* AMBIGUOUS REFERENCE TO DUPLICATED ENTRY - *epname* FROM MODULE - *module*
Two or more links exist between *epname* and *modname*. Design the segment structure and module DUP assignments to resolve the ambiguity.

*WARNING* 'BIN' FILE- *dn* IS A GLOBAL BINARY INPUT FILE
You have designated the binary input file *dn* to be both local and global. Correct the file to be just one or the other.

*ERROR* 'BIN' FILE- *dn* IS FORCE-LOADED INTO SEGMENT- *segname*
You have previously assigned dataset *dn* to segment *segname*. Assign *dn* to at most one segment.

*ERROR* BINARY DATASET *dn* DOES NOT BEGIN WITH A 'PDT'
Dataset *dn* contains invalid data. Check for incorrect spelling of the dataset name or ensure that the dataset was generated correctly.

*CAUTION* BLANK COMMON BLOCK DATA-LOAD BY MODULE- *modname*
DATA-LOADING SKIPPED
The user code attempts to preload the blank common block, which is illegal. Remove initialization presets for data in blank common.

*WARNING* CANNOT FIX TASK COMMON BLOCK - *cbname* TO NON-ROOT SEGMENT - *segname*
The common block *cbname* is a task common block and was assign to a segment. Remove the task common block name from the COMMONS directive, or make *cbname* not a task common block.

*CAUTION* CODE OR LOCAL DATA LOADED AT ADDRESS GREATER THAN 4MWD
The program has a module loaded at an address greater than 4 million words. Code cannot execute correctly if loaded above this address. Data can only be accessed correctly if special coding instructions have been used. You should move data to common blocks or make the size of local data areas smaller.

*WARNING* COMMON BLOCK - *cbname* REDEFINED AS A TASK COMMON BY MODULE - *modname*
The common block *cbname* was encountered in another module and was a regular common block, while in module *modname* the common block was a task common. Change the code so that all references to the same common block are consistant.

*CAUTION* COMMON BLOCK- *cbname* NOT REFERENCED, DISCARDED
No references are made to *cbname* and it is discarded. If you intend to reference *cbname*, check the spelling.

*WARNING* COMPILATION ERRORS IN MODULE- *modname* IN DATASET- *dn*
The user code contains compilation errors. Locate and correct the errors.

*ERROR* DATASET - *dn* CANNOT BE FOUND
SEGLDR cannot find dataset *dn*. Access or create dataset *dn* and rerun.

*CAUTION* DUPLICATE ENTRY- *epname* DISCOVERED IN FILE- *dn*; IGNORED
Two or more copies of *epname* exist. Any subsequent occurrences found are ignored.

*CAUTION* DUPLICATE MODULE- *modname* IN DATASET *dn* ENCOUNTERED AND IGNORED

This message means that *modname* appears more than once in datasets specified by the BIN directive. The extra copies are ignored.

*CAUTION* DYNAMIC COMMON BLOCK- *cbname* DATA-LOADED BY MODULE- *modname* DATA-LOADING SKIPPED

The user code attempts to preload the dynamic common block *cbname*, which is illegal. Remove initialization presets for data in common block *cbname*.

*ERROR* DYNAMIC COMMON BLOCK- *cbname* NOT LOADED

The common block named by the DYNAMIC directive is not found in any user code. Include the common block in the code or remove the directive.

*NOTE* 'ENDSEG' DIRECTIVE MISSING, ASSUMED

Two SEGMENT directives were found without an intervening ENDSEG directive or an end-of-file was read while reading a segment descrption. Insert the ENDSEG directive before beginning a new set of segment descriptions.

*WARNING* ENTRY *epname1* IS A SYNONYM OF- *epname2*

*epname1* is redefined in an EQUIV directive and is also included in MODULES directive. Remove *epname1* from either the EQUIV directive or the MODULES directive.

*CAUTION* ENTRY- *epname* IS IN FORCE-LOAD FILE- *dn* BUT WILL BE LOADED FROM FILE- *dn*

*epname* appears in the BIN load file *dn* and in a file specified in a MODULES directive. *epname* is loaded from the file named on the MODULES directive.

*CAUTION* ENTRY- *epname* UNUSED; DISCARDED

Because no references are made to *epname*, it is discarded. If you intend to reference *epname*, check the spelling.

*WARNING* ENTRY- *epname* DUPLICATED BUT NOT NAMED IN A 'DUP' DIRECTIVE FOR SEGMENT- *segname*

More than one entry of *epname* exists. Use the DUP directive to specify *epname* in all segments where *epname* is required.

*WARNING* ENTRY- *epname* HAS BEEN PREVIOUSLY DEFINED

*epname* is already the target entry point of the synonym list in another EQUIV directive. EQUIV redefinitions cannot be nested. Remove *epname* from this synonym list or correct the earlier EQUIV directive's target entry.

*WARNING* ENTRY- *epname* PREVIOUSLY REDEFINED AS A SYNONYM

*epname* is already used as a synonym in an earlier EQUIV directive. EQUIV redefinitions cannot be nested. Remove *epname* from earlier synonym list or change this EQUIV directive's target entry point.

*WARNING* ENTRY- *epname1* IS ALREADY A SYNONYM OF ENTRY- *epname2*

You have used the EQUIV directive to define *epname1* as a synonym when *epname1* has already been defined as a synonym of *epname2*. Rewrite EQUIV directives so that *epname1* is a synonym for at most one other symbol.

*CAUTION* GARBAGE AFTER ENDSEG IGNORED

Extraneous characters follow the ENDSEG directive. Remove the extraneous characters.

*WARNING* GARBAGE AFTER NODEFLIB- *symbol*

An unexpected symbol follows the keyword NODEFLIB. Change the symbol to a semicolon (;) or remove extra characters.

*WARNING* ILLEGAL '*dir*' VALUE - *value*

The *dir* directive has a value containing a non-numeric character, or the value is not valid for the directive. Correct the directive's value.

*WARNING* ILLEGAL REFERENCE TO COMMON BLOCK- *cbname* IN SEGMENT- *segname* FROM MODULE- *modname* IN SEGMENT *segment*

Module *modname* references common block *cbname* but the segments containing each are not able to be in memory at the same time. Design the segment structure or module assignments or specify a copy of *cbname* as a segment accessible to the module.

*WARNING* ILLEGAL REFERENCE TO ENTRY - *epname* IN SEGMENT - *segname1* FROM MODULE - *modname* IN SEGMENT - *segname2*

MODULE *modname* references entry point *epname* but the segments containing each are not able to be in memory at the same time. Design the segment structure or module assignments of use the DUP directive to provide an accessible entry point to the module.

*CAUTION* INITIAL MANAGED MEMORY TOO SMALL-INCREASED TO MINIMUM

The initial size of managed memory is not large enough to include the stack and necessary overhead, and it cannot expand. Specify a larger initial heap size or do not fix the heap size.

*CAUTION* INTER-SEGMENT DATA-LOAD TO COMMON BLOCK- *cbname* FROM MODULE- *modname* DATA LOADING SKIPPED

An attempt was made to preset common variables with a DATA statement in a module that is in a different segment than the one in which the common block is assigned. If data loading is needed, put the DATA statement in a module in the same segment as *cbname*.

*CAUTION* LENGTH OF COMMON BLOCK - *cbname* REDEFINED BY MODULE - *modname*
The user code in *modname* defines *cbname* as using more or less memory than was originally allocated to *cbname*. Make all references to *cbname* consistent (see the REDEF directive ).

*WARNING* LENGTH OF COMMON BLOCK- *cbname* REDEFINED BY MODULE- *modname*
The user code in *modname* defines *cbname* as using less or more memory than was originally allocated to *cbname*. Make all references to *cbname* consistent (see the REDEF directive).

*CAUTION* MEMORY EPSILON VALUE TOO SMALL-INCREASED TO MINIMUM
The value specified as the heap *epsilon* value was below the minimum acceptable. Specify a value of 2 or greater.

*WARNING* MISPLACED '*dir*' DIRECTIVE
The *dir* directive appears outside SEGMENT/ENDSEG directive group. Place the *dir* directive inside the appropriate SEGMENT/ENDSEG directive group or groups.

*NOTE* MODULE - *modname* AT ADDRESS _ *addr* CONTAINS A RELOCATABLE FIELD 22 BITS LONG -- CANNOT RUN IN EMA MODE
The reference at address *addr* is exactly 22 bits long. If the program were to be run with EMA mode enabled, the hardware would treat this large address as a negative number. If you wish to run with EMA mode enabled, you must change these reverences. See a Cray Research site analyst.

*CAUTION* MODULE- *modname* IN DATASET - *dn* HAS NO ENTRIES; MODULE SKIPPED
*modname* has no entry points. If you want to include *modname*, place an entry point in the code and reference it in an included module.

*CAUTION* MODULE- *modname* IN DATASET - *dn* IS A RELOCATABLE OVERLAY; MODULE SKIPPED
*modname* is an operating system relocatable overlay. *modname* cannot be loaded and is skipped. Remove *modname* from dataset, or do not include dataset *dn*.

*WARNING* MODULE- *modname* IS ALREADY FIXED AND IS A SYNONYM
*modname* appears in an EQUIV directive, but is assigned to a segment by the MODULES directive. Remove *modname* from either the EQUIV or MODULES directives.

*WARNING* MORE DIRECTIVES EXPECTED - LOOK FOR EMBEDDED EOF
The end of the directive file was encountered before the ENDTREE directive and/or segment descriptions were found. Include the ENDTREE directive and the segment description in the segment tree definition directives. Check for end-of-file record embedded within the directive file.

*WARNING* MORE THAN ONE ROOT SEGMENT IN TREE - *segname*
*segname* is one of two or more segments with no predecessor. Define a tree with exactly one root.

*WARNING* MULTI-TASKING LIBRARIES NOT USED
Ther is at lest one task common block reference in the load, but the routine which allocates task common blocks at execution time was not loaded. Be sure that the libraries are available. See a Cray site analyst.

*WARNING* NO LEGAL LINK FROM MODULE- *modname* TO DUPLICATED COMMON BLOCK- *cbname*
No copy of common block *cbname* is accessible to *modname*. Rewrite the module assignments or provide another copy of *cbname* to provide a valid link.

*WARNING* NO LEGAL LINK FROM MODULE- *modname* TO DUPLICATED ENTRY - *epname*
No copy of the module containing *epname* is accessible to *modname*. Rewrite the module assignments to provide a valid link.

*CAUTION* NO SYMBOL TABLE INFOMATION FOR USE WITH SID
Both the SID and SYMBOLS=OFF directives were specified. No symbol table information will be written for use with SID. Remove either the SID or SYMBOLS=OFF directive.

*WARNING* NO ROOT SEGMENT IN TREE
(1) Every segment in the tree description has a predecessor. Correct the segment tree description. (2) No tree description occurs between the TREE and ENDTREE directives. Insert the segment tree description.

*ERROR* NO TRANSFER ENTRY POINT
No transfer entry point was specified with the XFER directive and no primary entry point exists to which control can be given to begin execution. Use the XFER directive to specify the desired initial entry point of the program.

*WARNING* NO VALUE ASSIGNED TO- *symbol*
Nothing follows the '=' after the keyword *symbol*. Insert the desired value after the '='.

*CAUTION* NON-NUMERIC MANAGED MEMORY INCREMENT-DEFAULT USED
The heap increment value contains a non-numeric character. Specify a numeric value.

*CAUTION* NON-NUMERIC MANAGED MEMORY SIZE-DEFAULT USED
The heap size value contains a non-numeric character. Specify a numeric value.

*CAUTION* NON-NUMERIC MEMORY EPSILON VALUE-DEFAULT USED
The heap *min* value contains a non-numeric character. Specify a numeric value.

*CAUTION* NON-NUMERIC STACK INCREMENT-DEFAULT USED
The stack increment value contains a non-numeric character. Specify a numeric value.

*CAUTION* NON-NUMERIC STACK SIZE-DEFAULT USED
The stack size value contains a non-numeric character. Specify a numeric value.

*WARNING* PREMATURE END OF '*dir*' DIRECTIVE
A comma at the end of a line is followed by an end-of-file instead of a record. Use the correct form of the *dir* directive, complete the list to be processed or remove the trailing comma. Check for an embedded end-of-file record.

*WARNING* PREMATURE END OF SUCCESSOR LIST- *segname*
An end of line appears immediately after a left parenthesis or the successor list is not completed with a right parenthesis. Complete the successor list on one line or end the line with a comma to continue the list on the next line.

*WARNING* 'PRESET' PATTERN *value* EXCEEDS 16 BITS
The PRESET directive names a value too large to be represented by 16 binary digits. Change the value so that it is within range.

*CAUTION* PROGRAM CANNOT RUN IN EM MODE BECAUSE MODULE - *modname*
REFERENCES *number* ADDRESSES BETWEEN 2MWD AND 4MWD
The module named ahs references that would not be made correctly if run with EMA mode enabled on a machine with greater than 4 million words. Changing the MLEVEL print level to NOTE will cause all occurances of these reference types to be listed. See a Cray Research analyst to get a complete discussion of EMA usages.

*ERROR* READING BINARY FILE- *dn* RECORD NUMBER- *value*
A read error occurred in the dataset *dn*. Rerun the job. Recreate the dataset and run the job again. If the job still fails, see a Cray Research site analyst.

*CAUTION* REFERENCE AT ADDRESS- *addr* IN MODULE- *modname* HAS PARCEL ATTRIBUTE
ENTRY- *epname* HAS WORD ATTRIBUTE
EXTERNAL REFERENCE LINKED TO PARCEL ZERO
The entry point *epname* is a data word, but the reference in module *modname* treats the entry like an instruction. Change the code to ensure that *epname* is used consistently.

*CAUTION* REFERENCE AT ADDRESS- *addr* IN MODULE- *name* HAS WORD ATTRIBUTE ENTRY- *epname* HAS PARCEL ATTRIBUTE
EXTERNAL REFERENCE LINKED AS A WORD ADDRESS

The entry point *epname* is an instruction, but module *modname* refers to *epname* as a data word. Change the code to ensure that *epname* is used consistently.

*CAUTION* REFERENCE TO COMMON BLOCK- *cbname* IN SEGMENT- *segname* FROM MODULE- *modname* IN SEGMENT- *segname* IS UNSAFE

*cbname* may not be in memory when module *modname* is executed. Verify that *cbname* would be in memory when referenced. If it is not, rewrite the module or common block assignments.

*WARNING* RELOCATION FIELD OVERFLOW IN MODULE- *modname* AT ADDRESS *address*

The value to be loaded at address *address* is too large or too small to fit into the field. Check the expression calculation.

*WARNING* RESERVED OR ILLEGAL '*dir*' DATASET NAME- *dn*

Dataset name *dn* on the *dir* directive does not follow COS naming conventions or is reserved by SEGLDR. Change *dn* to a valid name.

*WARNING* SECONDARY ENTRY- *epname* IN MODULE- *modname* NOT DECLARED BY A 'DUP' DIRECTIVE

A secondary entry point in a duplicated module is not declared as a duplicate. Include *epname* in a DUP directive.

*WARNING* SEGLDR RESIDENT ROUTINE NOT FOUND

The routine that handles intersegment subroutine calls was not located. Normally, this subroutine appears in one of the default libraries. Possible causes of this error are the following:

- The NODEFLIB directive was used and a SEGLDR resident routine is not provided by the user.
- The resident routine is not provided in a system-supplied library (see the Cray Research site analyst).

*WARNING* SEGMENT- *segname* HAS BEEN PREVIOUSLY DEFINED

The segment tree description directives already defined *segname*'s successor segments or defined *segname* as a successor of another segment. Correct or remove those directives that cause multiple definitions.

*WARNING* SEGMENT- *segname* IS EMPTY

*segname* has no modules named by MODULES or BIN directives, or the modules named were not referenced and were trimmed. Every segment must contain at least one module. Provide *segname* with at least one module or remove the segment.

*WARNING* SEGMENT- *segname* IS UNDEFINED

A segment in a DUP list is not defined in the segment tree. Include *segname* in the segment tree description directives or remove it from the DUP list.

*WARNING* SEGMENT- *segname* WAS NOT DESCRIBED BY A 'SEGMENT' DIRECTIVE

An end-of-file was reached before a SEGMENT/ENDSEG group describing *segname* was found. Include a SEGMENT/ENDSEG group describing *segname* or remove all references to *segname*.

*WARNING* SEGMENT-*segname* NOT INCLUDED IN TREE DESCRIPTION

*segname* is not defined in the segment tree. Include *segname* in the segment tree description directives, or remove *segname*'s segment description.

*WARNING* SEPARATOR EXPECTED AFTER- *symbol*

There are unexpected characters after *symbol*. Remove the extra characters or insert the directive separator ';'.

*CAUTION* STACK TOO SMALL-INCREASED TO MINIMUM

The value specified as the initial stack size was below the minimum acceptable. Specify a value of 128 or greater.

*WARNING* SUCCESSOR=PREDECESSOR

A directive caused a segment in the segment tree to be defined as its own successor. Correct the segment tree description directive.

*CAUTION* TASK COMMON BLOCK - *cbname* DATA-LOADED BY MODULE - *modname*

The user code attempts to preload the task common block, which is illegal. Remove initialization presets for data in common block named.

*WARNING* TASK COMMON BLOCK - *cbname* REDEFINED AS A REGULAR COMMON BY MODULE - *modname*

The common block name was encountered in another module and was a task common block; while in module *modname*, the common block was a regular common. Change code so that all references to the same common block are consistant.

*WARNING* TOO MANY SEGMENT CALLS-- INCREASE SLT ; ACTUAL SLT REQUIREMENT- *value*

Too little space exists in the Segment Linkage Entry Table for callers and callees. Use the SLT directive to increase the Segment Linkage Entry Table to the required size as stated.

*WARNING* TOO MANY SEGMENTS

The 1000-segment limit was exceeded. Reduce the number of segments.

*ERROR* TRANSFER ENTRY POINT - *epname* IS NOT THE ROOT SEGMENT

The entry point *epname* is not in the root segment and is specified as the entry point for beginning execution. Put the transfer entry point in the root segment or specify another entry point within the root segment as the transfer entry point using the XFER directive.

*WARNING* UNFIXED MODULE- *modname* REFERENCES DUPLICATED COMMON BLOCK- *cbname*

Duplicate common blocks were specified but not all modules which reference them are fixed. Fix the modules that reference duplicate common blocks.

*WARNING* UNFIXED MODULE *modname* REFERENCES DUPLICATED ENTRY - *epname*

The module is not assigned to a module and references the duplicate entry point *epname*. Use the MODULES directive to fix the module in the segment desired.

*WARNING* UNKNOWN '*dir*' KEYWORD - *value*

The value named is not a valid keyword for the '*dir*' directive. Use the correct keyword.

*WARNING* UNKNOWN OR MISPLACED DIRECTIVE - *symbol*

The *symbol* directive is not recognized as a SEGLDR directive or is not valid at this point in the directives. Check spelling and delimiters or move the directive to the correct place in directive sequence.

*ERROR* 'XFER' ENTRY POINT - *epname* NOT FOUND

*epname* was not found. Include *epname* in the code. Check the spelling.

# MAPPING G

This section provides a set of SEGLDR directives, block maps and associated output, and related entry point and common block reference maps, for the example FORTRAN program given below. The following FORTRAN program consists of ten subroutines. Each module is loaded into a separate segment.

EXAMPLE FORTRAN PROGRAM

```
      PROGRAM EXAMPLE
      DATA I /0/
      CALL SUBR1(I)
      CALL SUBR2(I)
      PRINT *,' VALUE OF I IS ',I
      END

      SUBROUTINE SUBR1(I)
      COMMON /SPACE/ SPACE(100)
      COMMON COMMON
      I=I+1
      CALL SUBR1A(I)
      CALL SUBR1B(I)
      CALL SUBR1C(I)
      RETURN
      END

      SUBROUTINE SUBR1A(I)
      COMMON COMMON
      PRINT *,' EXECUTION OF SUBR1A'
      I=I+1
      RETURN
      END

      SUBROUTINE SUBR1B(I)
      COMMON /STATUS/ STATUS
      COMMON COMMON
      PRINT *,' EXECUTION OF SUBR1B'
      I=I+1
      RETURN
      END
```

```
SUBROUTINE SUBR1C(I)
COMMON /STATUS/ STATUS
PRINT *,' EXECUTION OF SUBR1C'
I=I+1
RETURN
END

SUBROUTINE SUBR2(I)
COMMON /SPACE/ SPACE(100)
I=I+1
CALL SUBR2A(I)
RETURN
END

SUBROUTINE SUBR2A(I)
PRINT *,' EXECUTION OF SUBR2A'
I=I+1
CALL SUBR2B(I)
RETURN
END

SUBROUTINE SUBR2B(I)
PRINT *,' EXECUTION OF SUBR2B'

I=I+1
CALL SUBR2C(I)
RETURN
END

SUBROUTINE SUBR2C(I)
PRINT *,' EXECUTION OF SUBR2C'
I=I+1
CALL SUBR2D(I)
RETURN
END

SUBROUTINE SUBR2D(I)
PRINT *,' EXECUTION OF SUBR2D'
I=I+1
RETURN
END
```

## SEGLDR DIRECTIVES FOR SAMPLE PROGRAM

The following sample SEGLDR directive input is used to specify and diagram the construction of the segmented object module.

```
MAP=FULL
DYNAMIC=//
ABS=EXAMPLE
*************************************************
*                                               *
*                        ROOT                   *
*                       /                       *
*                      /                        *
*                     /                         *
*                    /                          *
*                   /                           *
*                  SEG1          SEG2           *
*                 / |             |             *
*                /  |             |             *
*               /   |           SEG2A           *
*              /    |             |             *
*          SEG1A SEG1B SEG1C      |             *
*                                SEG2B          *
*                                  |            *
*                                  |            *
*                                SEG2C          *
*                                  |            *
*                                  |            *
*                                SEG2D          *
*                                               *
*************************************************
TREE
  ROOT(SEG1,SEG2)
  SEG1(SEG1A,SEG1B,SEG1C)
  SEG2(SEG2A)
  SEG2A(SEG2B)
  SEG2B(SEG2C)
  SEG2C(SEG2D)
ENDTREE
SEGMENT=ROOT
  MODULES=EXAMPLE
ENDSEG
*
*  Left-hand segment tree branch
*
SEGMENT=SEG1
  MODULES=SUBR1
ENDSEG
SEGMENT=SEG1A
  MODULES=SUBR1A
ENDSEG
SEGMENT=SEG1B
  MODULES=SUBR1B
ENDSEG
```

```
SEGMENT=SEG1C
  MODULES=SUBR1C
ENDSEG
*
*  Right-hand segment tree branch
*
SEGMENT=SEG2
  MODULES=SUBR2
ENDSEG
SEGMENT=SEG2A
  MODULES=SUBR2A
ENDSEG
SEGMENT=SEG2B
  MODULES=SUBR2B
ENDSEG
SEGMENT=SEG2C
  MODULES=SUBR2C
ENDSEG
SEGMENT=SEG2D
  MODULES=SUBR2D
ENDSEG
```

## EXAMPLE SEGLDR MAP OUTPUT FOR SAMPLE PROGRAM

The sample SEGLDR output following is an example of the general information preceding the block maps. Note that word addresses and block lengths are in octal.

```
PROGRAM STATISTICS

 SEGMENTED OBJECT MODULE WRITTEN TO- EXAMPLE
 PROGRAM ORIGIN-      200
 MAXIMUM SEGMENT CHAIN LENGTH-    22647 WORDS, ENDING WITH SEGMENT- SEG2D
 TRANSFER IS TO ENTRY POINT- EXAMPLE  AT ADDRESS-    12613a
 DYNAMIC COMMON BLOCK- //
         ORIGIN-    22647
         LENGTH-        1
 ACTUAL SLT REQUIREMENT-       16
```

## SAMPLE PROGRAM BLOCK MAPS

There is one block map for each segment of the program in this example. Block map 1 is an abrreviated sample because some library routines were ommitted for readability.

```
BLOCK MAP 1

SEGMENT- ROOT     ORIGIN-      200 LENGTH-    22213 SAVE=OFF
ROOT SEGMENT
SORTED BY ASCENDING BLOCK ORIGIN ADDRESS

NAME        ADDRESS   LENGTH  SOURCE       DATE        OS REV       PROCESSOR      COMMENT

/$SEGRES /      200      445
/SPACE   /      645      144
$SEGRES        1011    11700  $UTLIB   09/13/82 08:42  COS 1.12  CAL 1.12 09/04/82
EXAMPLE       12711       67  $BLD     09/16/82 08:53  COS 1.12  CFT 1.11 09/04/82
**USX**       13000      134           09/16/82 08:53  COS 1.12   SEGLDR  1.12
$MEMUC20      13134      276  $SYSLIB  09/15/82 02:48  COS 1.12  CAL 1.12 09/04/82

SORTED BY BLOCK NAME

NAME        ADDRESS   LENGTH  SOURCE       DATE        OS REV     PROCESSOR        COMMENT

$MEMUC20      13134      276  $SYSLIB  09/15/82 02:48  COS 1.12  CAL 1.12 09/04/82
$SEGRES        1011    11700  $UTLIB   09/13/82 08:42  COS 1.12  CAL 1.12 09/04/82
/$SEGRES /      200      445
**USX**       13000      134           09/16/82 08:53  COS 1.12   SEGLDR  1.12
EXAMPLE       12711       67  $BLD     09/16/82 08:53  COS 1.12  CFT 1.11 09/04/82
/SPACE   /      645      144


BLOCK MAP 2

SEGMENT- SEG1     ORIGIN-    13432 LENGTH-       36 SAVE=OFF
IMMEDIATE PREDECESSOR SEGMENT- ROOT
SORTED BY ASCENDING BLOCK ORIGIN ADDRESS
 NAME        ADDRESS   LENGTH  SOURCE       DATE        OS REV       PROCESSOR      COMMENT
/STATUS  /     13432        1
 SUBR1         13433       35  $BLD     09/16/82 08:53  COS 1.12  CFT 1.11 09/04/82
SORTED BY BLOCK NAME
 NAME        ADDRESS   LENGTH  SOURCE       DATE        OS REV     PROCESSOR        COMMENT
/STATUS  /     13432        1
 SUBR1         13433       35  $BLD     09/16/82 08:53  COS 1.12  CFT 1.11 09/04/82


BLOCK MAP 3

SEGMENT- SEG2     ORIGIN-    13432 LENGTH-       24 SAVE=OFF
IMMEDIATE PREDECESSOR SEGMENT- ROOT
SORTED BY ASCENDING BLOCK ORIGIN ADDRESS
 NAME        ADDRESS   LENGTH  SOURCE       DATE        OS REV       PROCESSOR      COMMENT
 SUBR2         13432       24  $BLD     09/16/82 08:53  COS 1.12  CFT 1.11 09/04/82


BLOCK MAP 4

SEGMENT- SEG1A    ORIGIN-    13470 LENGTH-       57 SAVE=OFF
IMMEDIATE PREDECESSOR SEGMENT- SEG1
SORTED BY ASCENDING BLOCK ORIGIN ADDRESS
 NAME        ADDRESS   LENGTH  SOURCE       DATE        OS REV       PROCESSOR      COMMENT
 SUBR1A        13470       57  $BLD     09/16/82 08:53  COS 1.12  CFT 1.11 09/04/82
```

```
BLOCK MAP 5

SEGMENT- SEG1B    ORIGIN-    13470 LENGTH-       57 SAVE=OFF
IMMEDIATE PREDECESSOR SEGMENT- SEG1
SORTED BY ASCENDING BLOCK ORIGIN ADDRESS
 NAME       ADDRESS   LENGTH  SOURCE        DATE        OS REV       PROCESSOR      COMMENT
 SUBR1B       13470       57  $BLD     09/16/82 08:53  COS 1.12  CFT 1.11 09/04/82


BLOCK MAP 6

SEGMENT- SEG1C    ORIGIN-    13470 LENGTH-       57 SAVE=OFF
IMMEDIATE PREDECESSOR SEGMENT- SEG1
SORTED BY ASCENDING BLOCK ORIGIN ADDRESS
 NAME       ADDRESS   LENGTH  SOURCE        DATE        OS REV       PROCESSOR      COMMENT
 SUBR1C       13470       57  $BLD     09/16/82 08:53  COS 1.12  CFT 1.11 09/04/82


BLOCK MAP 7

SEGMENT- SEG2A    ORIGIN-    13456 LENGTH-       63 SAVE=OFF
IMMEDIATE PREDECESSOR SEGMENT- SEG2
SORTED BY ASCENDING BLOCK ORIGIN ADDRESS
 NAME       ADDRESS   LENGTH  SOURCE        DATE        OS REV       PROCESSOR      COMMENT
 SUBR2A       13456       63  $BLD     09/16/82 08:53  COS 1.12  CFT 1.11 09/04/82


BLOCK MAP 8

SEGMENT- SEG2B    ORIGIN-    13541 LENGTH-       63 SAVE=OFF
IMMEDIATE PREDECESSOR SEGMENT- SEG2A
SORTED BY ASCENDING BLOCK ORIGIN ADDRESS
 NAME       ADDRESS   LENGTH  SOURCE        DATE        OS REV       PROCESSOR      COMMENT
 SUBR2B       13541       63  $BLD     09/16/82 08:53  COS 1.12  CFT 1.11 09/04/82


BLOCK MAP 9

SEGMENT- SEG2C    ORIGIN-    13624 LENGTH-       63 SAVE=OFF
IMMEDIATE PREDECESSOR SEGMENT- SEG2B
SORTED BY ASCENDING BLOCK ORIGIN ADDRESS
 NAME       ADDRESS   LENGTH  SOURCE        DATE        OS REV       PROCESSOR      COMMENT
 SUBR2C       13624       63  $BLD     09/16/82 08:53  COS 1.12  CFT 1.11 09/04/82


BLOCK MAP 10

SEGMENT- SEG2D    ORIGIN-    13707 LENGTH-       57 SAVE=OFF
IMMEDIATE PREDECESSOR SEGMENT- SEG2C
SORTED BY ASCENDING BLOCK ORIGIN ADDRESS
 NAME       ADDRESS   LENGTH  SOURCE        DATE        OS REV       PROCESSOR      COMMENT
 SUBR2D       13707       57  $BLD     09/16/82 08:53  COS 1.12  CFT 1.11 09/04/82
```

## SAMPLE PROGRAM ENTRY POINT CROSS-REFERENCE MAP

This sample entry point cross-reference map shows entry point values (addresses assigned), segments to which modules are assigned, and the segment tree in caller/callee form. As for block map 1, this sample is abbreviated for readability.

```
MODULE      ENTRY       VALUE          SEGMENT
$MEMUC20    $MEMUC20    13347a         ROOT
                                         CALLED BY...  $SEGRES
            $MEMUC50    13357a           CALLED BY...
$SEGRES     $SEGRES      1011a         ROOT    CALLS...  $MEMUC20
                                             CALLED BY...  EXAMPLE

            $SEGCALL     2234a
**USX**     $WLI        13100a         ROOT
                                         CALLED BY...  EXAMPLE    SUBR1A   SUBR1B   SUBR1C   SUBR2A
                                                       SUBR2B     SUBR2C   SUBR2D
            $WLA        13101a           CALLED BY...  EXAMPLE    SUBR1A   SUBR1B   SUBR1C   SUBR2A
                                                       SUBR2B     SUBR2C   SUBR2D
            $WLV%       13102a           CALLED BY...  EXAMPLE
            $WLF        13103a           CALLED BY...  EXAMPLE    SUBR1A   SUBR1B   SUBR1C   SUBR2A
                                                       SUBR2B     SUBR2C   SUBR2D
            $END        13104a           CALLED BY...  EXAMPLE
            $BTD        13105a           CALLED BY...
            $BTO        13106a           CALLED BY...

EXAMPLE     EXAMPLE     12721a         ROOT    CALLS...  SUBR1      SUBR2    $WLI     $WLA     $WLV%
                                                         $WLF       $END     $SEGRES

SUBR1       SUBR1       13437a         SEG1      CALLS...  SUBR1A     SUBR1B   SUBR1C
                                           CALLED BY...  EXAMPLE
SUBR1A      SUBR1A      13500a         SEG1A     CALLS...  $WLI       $WLA     $WLF
                                           CALLED BY...  SUBR1
SUBR1B      SUBR1B      13500a         SEG1B     CALLS...  $WLI       $WLA     $WLF
                                           CALLED BY...  SUBR1
SUBR1C      SUBR1C      13500a         SEG1C     CALLS...  $WLI       $WLA     $WLF
                                           CALLED BY...  SUBR1
SUBR2       SUBR2       13436a         SEG2      CALLS...  SUBR2A
                                           CALLED BY...  EXAMPLE
SUBR2A      SUBR2A      13466a         SEG2A     CALLS...  $WLI       $WLA     $WLF     SUBR2B
                                           CALLED BY...  SUBR2
SUBR2B      SUBR2B      13551a         SEG2B     CALLS...  $WLI       $WLA     $WLF     SUBR2C
                                           CALLED BY...  SUBR2A
SUBR2C      SUBR2C      13634a         SEG2C     CALLS...  $WLI       $WLA     $WLF     SUBR2D
                                           CALLED BY...  SUBR2B
SUBR2D      SUBR2D      13717a         SEG2D     CALLS...  $WLI       $WLA     $WLF
                                           CALLED BY...  SUBR2C
```

## SAMPLE PROGRAM COMMON BLOCK REFERENCE MAP

Like the other maps in this appendix, this sample output is based on the example FORTRAN program included at the start of this section.

COMMON BLOCK REFERENCES

```
BLOCK    SEGMENT    ADDRESS   LENGTH   MODULE REFERENCES

$SEGRES  ROOT           200      445   $SEGRES
//       SEG1         27420        1   SUBR1      SUBR1A     SUBR1B
SPACE    ROOT           645      144   SUBR1      SUBR2
STATUS   SEG1         13432        1   SUBR1B     SUBR1C
```

# GLOSSARY

# GLOSSARY

B

Binary input dataset - A dataset consisting of relocatable modules produced by a language processor. Each relocatable module is contained in a single record. Only the first file of binary input datasets is examined by SEGLDR. There are three types of binary input datasets: Global BIN datasets, segment description BIN datasets, and library datasets.

Block - A general term describing either a module or common block.

Branch segment - Any nonroot segment.

D

Data loading - The process by which SEGLDR inserts data into object module blocks. Data loading occurs explicitly in response to program DATA statements that name common block variables. Implicit data loading of locations within a subprogram code block can also occur if the compiler or assembler so dictate.

E

Entry point - A symbolic name defined by a language processor to be a starting address for execution within a module. CFT PROGRAM, FUNCTION, ENTRY and SUBROUTINE statements name entry points. CAL allows entry point names to be absolute values and data items as well. The following CAL code fragment identifies entry point VALUE as an absolute value and entry point DATA as a data item.

```
          ENTRY     VALUE,DATA
VALUE     =         D'64
DATA      DATA      'STRING'
```

F

Fixed - Assigned to a particular segment of the tree.

Force-loading - Inclusion of an entry point that has no callers. Force-loading is performed on BLOCKDATA modules, for example. Force-loading of all uncalled entry points can be enabled by the FORCE directive.

M

Module - A subprogram. For CFT, a module name is defined by the PROGRAM, BLOCKDATA, FUNCTION, or SUBROUTINE statement. CAL modules are named by the IDENT pseudo-op.

Movable block - A module or common block not assigned by a segment description directive to a specific segment but assigned by SEGLDR to the highest-level segment preceding all callers.

O

Object module - The executable binary program produced by SEGDLR. Formatted so that it can be read to memory by COS for execution, it is written to the first file of the ABS dataset.

P

Primary entry point - An entry point named by the CFT PROGRAM statement or the CAL START pseudo-op which serves as the default transfer address for the program. (If there is no PROGRAM, SUBROUTINE, FUNCTION, or BLOCKDATA statement, CFT substitutes PROGRAM $MAIN.) The first primary entry point encountered is the default transfer address.

R

Root segment - The single segment occupying the root node of the segment tree. The root segment is always memory-resident during program execution.

S

Segment - A portion of code which may be overlayed during execution. Segments differ from traditional overlays in that no explicit call must be made to an overlay manager in order to read code segments to memory.

Segment datasets - Temporary datasets created by the resident routine ($SEGRES) as part of its initialization. Each segment dataset consists of a single record containing a segment.

T

Transfer address - The location within a module at which program execution begins. The transfer address is symbolically identified by an entry point name. If a transfer address is not named by the XFER directive, a *primary entry point* is required (see *primary entry point* in this glossary).

Tree trimming - The process by which SEGLDR eliminates uncalled entry points from the object module.

U

Unsatisfied external - A reference (for example, a subroutine call) to an unknown entry point. SEGLDR searches all user-specified BIN and LIB datasets and default libraries for a subprogram to which a linkage can be made. Should the search fail, the calling module is said to contain an *unsatisfied external*.

# INDEX

# INDEX

## READERS COMMENT FORM

Segment Loader (SEGLDR) Reference Manual SR-0066 A

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME _____

JOB TITLE _____

FIRM _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

Attention:
PUBLICATIONS

**1440 Northland Drive**
**Mendota Heights, MN 55120**
U.S.A.

STAPLE

## READERS COMMENT FORM

Segment Loader (SEGLDR) Reference Manual SR-0066 A

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME _____

JOB TITLE _____

FIRM _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

Attention:
PUBLICATIONS

**1440 Northland Drive**
**Mendota Heights, MN 55120**
U.S.A.

Cray Research, Inc.
Publications Department
1440 Northland Drive
Mendota Heights, MN 55120
612-452-6650
TLX 298444