Segment Loader (SEGLDR™) and ld Reference Manual

SR–0066 9.0

Cray Research, Inc.

Portions of this product may still be in development. The existence of those portions still in development is not a commitment of actual release or support by Cray Research, Inc. Cray Research, Inc. assumes no liability for any damages resulting from attempts to use any functionality or documentation not officially released and supported. If it is released, the final form and the time of official release and start of support is at the discretion of Cray Research, Inc.

The UNICOS operating system is derived from UNIX® System V. The UNICOS operating system is also based in part on the Fourth Berkeley Software Distribution (BSD) under license from The Regents of the University of California.

Requests for copies of Cray Research, Inc. publications should be sent to the following address:

Cray Research, Inc.
Distribution Center
2360 Pilot Knob Road
Mendota Heights, MN 55120
USA

Order desk +1–612–683–5907
Fax number +1–612–452–0141

New Features

Segment Loader (SEGLDR) and `ld` *Reference Manual* SR–0066 9.0

The 9.0 version of this manual removes references to CRAY-2 and CRAY X-MP systems. No new technical information has been added. In addition, the information on the massively parallel processing (MPP) system loader (MPPLDR) has been removed and is now located in the *Cray MPP Loader User's Guide*, publication SG–2514.

Cray Research Software Documentation Map

The illustration on the following pages highlights the major body of documentation available for Cray Research (CRI) customers. The illustration is organized into categories by audience designation:

Audience	Description
End users	Those who use the UNICOS operating system, products, applications, or networking software
Application and system programmers	Those who write or modify system or application code on a CRI system for the purpose of solving computer system, scientific, or engineering problems
System administrators	Those who perform system administration tasks, such as installation, configuration, and basic troubleshooting
System analysts	Those who perform advanced troubleshooting, tuning, and customization
Operators	Those who perform operational functions, such as performing system dumps, and those who administer an operator workstation

To use the map, find the audience designation closest to your specific needs or role as a CRI system user. Note that manuals under other audiences may also be of interest to you; manuals are listed only once, underneath the audience to which they most directly apply. Some manual titles are abbreviated. The date in the map's footer tells you when the information was last revised.

For more information

In addition to the illustration, you can use the following publications to find documentation specific to your needs:

- *Software Documentation Ready Reference,* publication SQ–2122, serves as a general index to the CRI documentation set. The booklet lists documents and man pages according to topic.
- *Software Overview for Users*, publication SG–2052, introduces the UNICOS operating system, its features, and its related products. It directs you to documentation containing user-level information.
- *User Publications Catalog*, publication CP–0099, briefly describes all CRI manuals available to you, including some not shown on the map, such as training workbooks and other supplementary documentation.

Ordering

To obtain CRI publications, order them by publication number from the Distribution Center:

Cray Research, Inc.
Distribution Center
2360 Pilot Knob Road
Mendota Heights, MN 55120
USA

Order desk (612) 683-5907
Fax number (612) 452-0141

END USERS

Introductory

Software Overview for Users (SG–2052)●★

User's Guide to Online Information (SG–2143)●★

General

Software Documentation Ready Reference (SQ–2122)★

User Commands Reference (SR–2011)▲

User Commands Ready Reference (SQ–2056)▲

Korn Shell Ready Reference (SQ–2115)

UNICOS Shells Ready Reference (SQ–2116)

UNICOS Environment Variables Ready Reference (SQ–2117)

UNICOS Index for Man Pages (SR–2049)

Visual Interfaces Guide (SG–3094)●★

Tape Subsystem Guide (SG–2051)●★

Security (MLS) Guide (SG–2111)●

MPP Software Guide (SG–2508)●★

CRL

CRL User's Guide (SG–2126)★

Networking

NQS Guide (SG–2105)●★

TCP/IP and OSI Network Guide (SG–2009)●★

FTA Guide (SG–2144)●★

Text Editing

Text Editors Primer (SG–2050)

`vi` Reference Card (SQ–2054)

`ed` Reference Card (SQ–2055)

MVS Link

RQS User's Guide (SG–2405)

UNIX Link

NQE User's Guide (SG–2148)●

NQE Ready Reference (SQ–2149)

Introducing NQE (IN–2153)●

VAX/VMS Link

RQS User's Guide (SV–3151)

● Available online with CrayDoc
★ Available online with Docview
▲ Man pages available with the `man` command

10/94

APPLICATION AND SYSTEM PROGRAMMERS

Ada

Cray Ada Reference (SR–3014)★

Cray Ada Programming Guide (SR–3082)★

C

Cray Standard C Reference (SR–2074)●★

Cray Standard C Ready Reference (SQ–2076)

Cray Standard C for MPP (SR–2506)●★

CAL for CRAY Y-MP and CRAY Y-MP C90

Reference (SR–3108)●

Symbolic Machine Instructions (SR–3109)

Ready Reference (SQ–3110)

UNICOS Macros and Opdefs (SR–2403)▲

Cray Assembler for MPP

CAM Reference (SR–2510)●★

FORTRAN 77

CF77 Ready Reference (SQ–3770)

CF77 Commands and Directives (SG–3771)●★

CF77 Fortran Reference (SR–3772)●★

CF77 Optimization Guide (SG–3773)★

CF77 Message Manual (SR–3774)

Cray MPP Fortran Reference (SR–2504)●★

Fortran 90

CF90 Commands and Directives (SR–3901)●★

CF90 Fortran Language Reference (SR–3902)●★

CF90 Ready Reference (SQ–3900)

Introducing CF90 SPARC Prog. Env. (IN–2155)●

Introducing DPE (IN–2163)●

Libraries

System Libraries (SR–2080)▲

System Libraries Ready Ref. (SQ–2147)▲

Scientific Libraries (SR–2081)▲

Math Library (SR–2138)▲

Application Programmer's I/O Guide (SG–2168)▲

Application Programmer Library Ref. Manual (SG–2165)▲

Introducing CrayLibs (IN–2167)●

PVM and HeNCE Ref. (SR–2501)●★

PVM Reference Card (SQ–2512)

Loaders

Loader Reference (SR–0066)●★

SEGLDR Ready Reference (SQ–0303)

Loader for MPP

Cray MPP Loader Guide (SG–2514)●

Networking

RPC Reference (SR–2089)●★

Kerberos User's Guide (SG–2409)●★

Programming Tools

UNICOS Message System Programmer's Guide (SG–2121)●★

Compiler Information File (CIF) Reference (SR–2401)●★

CDBX Debugger Reference (SR–2091)●★

CDBX Debugger User's Guide (SG–2094)●★

CDBX Reference Card (SQ–2110)

Program Browser (`xbrowse`) (IN–2140)●

Tuning Guide to Parallel Vector Applications (SG–2182)●

MPP Apprentice Tool (IN–2511)●

Introducing Cray TotalView Debugger (IN–2502)●

Simulators

Cray MPP Simulator Guide (SG–2503)●

Source Control

USM User's Guide (SG–2097)●★

System Calls

System Calls (SR–2012)▲

X Window System

Reference (SR–2101)●★

Ready Reference (SQ–2123)

OPERATORS

OWS-E/IOS-E

OWS-E/IOS-E Reference (SR–3077)▲

OWS-E/IOS-E Ready Reference (SQ–3080)

OWS-E/IOS-E Operator's Guide (SG–3078)

OWS-E/IOS-E Administrator's Guide (SG–3079)

● Available online with CrayDoc
★ Available online with Docview
▲ Man pages available with the `man` command

10/94

SYSTEM ADMINISTRATORS AND ANALYSTS

UNICOS

UNICOS Installation Guide (SG–2112)

Installation Ref. Card (SQ–2411)

UNICOS Installation Tool Menus and Help Files (SG–2412)

UNICOS System Administration (SG–2113)●★

Administrator Commands Reference (SR–2022)▲

Administrator Commands Ready Ref. (SQ–2413)▲

CRL

CRL Administrator’s Guide (SG–2127)★

DMF

DMF Administrator’s Guide (SG–2135)★

Security and Licensing

UNICOS System Security Overview (SG–2141)★

FLEXlm Guide (SG–2181)●★

UNICOS under UNICOS

UuU Administrator’s Guide (SG–2156)●★

CRAY EL Series

IOS Commands Reference (SR–2408)▲

IOS Commands Ready Ref. (SQ–2162)

UNICOS Basic Administration Guide (SG–2416)●★

UNICOS Installation Guide for CRAY Y-MP EL Systems (SG–5201)

IOS Messages (SQ–2402)

Networking

fy Driver Administrator’s Guide (SG–2132)

MPP

CRAY T3D Administrator’s Guide (SG–2507)●

MVS Link

RQS Administrator’s Guide (SG–2406)

VAX/VMS Link

RQS Administrator’s Guide (SV–3152)

UNIX Link

RQS Administrator’s Guide (SG–2120)

NQE Administration (SG–2150)●

NQE Installation (SG–5236)●

Analysts

File Formats and Special Files Reference (SR–2014)▲

Data Migration MSP Writer’s Guide (SN–2098)★

UNICOS Tuning Guide (SR–2099)●

UNICOS `nmake` Card (SQ–2146)

Installation and Configuration Tool Reference (SR–3090)

USCP

Front-end Protocol Internals (SM–0042)★

USCP Optimization (SN–2103)

● Available online with CrayDoc

★ Available online with Docview

▲ Man pages available with the `man` command

10/94

Record of Revision

The date of printing or software version number is indicated in the footer. Changes in rewrites are noted by revision bars along the margin of the page.

Version ***Description***

September 1983. Original printing.

01 December 1983. This change packet brings the manual into agreement with version 1.13 of the Cray operating system 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. This reprint obsoletes all previous versions.

B February 1986. This rewrite brings the manual into agreement with SEGLDR release 2.0 running under COS 1.15 and SEGLDR release 2.1 running under UNICOS 1.0. New material includes the NOECHO and TRIAL directives and related error messages. The manual has been reorganized: appendixes A, B, and C have become sections 8, 9, and 10, respectively; most of the directives in section 4 have been moved to section 3. This rewrite obsoletes all previous versions.

C September 1986. This reprint with revision brings the manual into agreement with SEGLDR release 3.1 running under COS version 1.16 and SEGLDR release 3.0 running under UNICOS version 2.0. New error messages have been added, obsolete ones have been removed, and the –V option has been added to the UNICOS segldr command line. Change bars in the left margin indicate technical changes. All trademarks are now documented in the record of revision. This reprint obsoletes all previous versions.

D June 1987. This reprint with revision brings the manual into agreement with SEGLDR release 4.0 running under UNICOS version 3.0. SEGLDR has a new option to its split directive and a new pair of directives for defining program calling sequences.

Version	***Description***
E	June 1988. This rewrite brings the manual into agreement with SEGLDR version 5.0, supporting the 4.0 release of UNICOS and the 1.17 release of COS. The UNICOS command line has 19 new options, and the COS control statement has 25 new parameters. Four new directives have been added, and parameters available to 6 existing directives have changed.
F	February 1989. This rewrite updates the manual to document SEGLDR version 5.1, supporting the 5.0 release of UNICOS. Additionally, the title of the manual has changed to reflect the fact that it now documents the UNICOS `ld`(1) command, which is an interface to the segment loader. SEGLDR 5.1 supports segmented programs on CRAY-2 systems. The UNICOS command line has two new options; four new directives have been added; and parameters for the `ORDER` directive have changed.
6.0	December 1990. This reprint with revision updates the manual to document SEGLDR version 6.0, supporting the UNICOS 6.0 release. COS support has been removed from SEGLDR version 6.0.
7.0	April 1992. This reprint with revision updates the manual to document SEGLDR version 7.0, supporting the UNICOS 7.0 release. For detailed descriptions of changes to SEGLDR and this manual, see the New Features page.
8.0	November 1993. This rewrite updates the manual to document SEGLDR version 8.0, supporting the UNICOS 8.0 release and the MPP system 1.0 release. For detailed descriptions of changes to SEGLDR and this manual, see the New Features page.
9.0	July 1995. This reprint removes references to CRAY-2 and CRAY X-MP, and the information related to the MPP loader.

Preface

This publication documents the segment loader (SEGLDR) release 9.0 on Cray PVP systems running under the Cray Research UNICOS 9.0 operating system. SEGLDR is a loader for segmented and nonsegmented programs produced by the following Cray Research assemblers and compilers:

- Cray Research Fortran CFT77 compiler
- Cray Research Fortran CF90 compiler
- Cray Pascal
- Cray C compiler
- Cray Standard C compiler
- Cray Ada

This reference manual describes the operation of the SEGLDR loader, method of code execution, common block use, and common block assignment. The glossary defines SEGLDR terminology. Readers are assumed to be experienced programmers who understand overlays and are familiar with loaders.

Additionally, this manual documents the UNICOS command `ld`, which is an interface to the segment loader in the style of traditional UNIX system loaders.

Related publications

Other Cray Research, Inc. (CRI) publications that you may find useful are as follows:

Language processor documentation:

- *Pascal Reference Manual*, publication SR–0060
- *Cray Standard C Reference Manual*, publication SR–2074
- *Cray Assembly Language (CAL) for Cray PVP Systems Reference Manual*, publication SR–3108

- *CF77 Fortran Language Reference Manual*, publication SR–3772
- *CF90 Fortran Language Reference Manual*, publication SR–3902

Operating system documentation:

- *UNICOS User Commands Reference Manual*, publication SR–2011

Library documentation:

- *UNICOS System Calls Reference Manual*, publication SR–2012
- *UNICOS System Libraries Reference Manual*, publication SR–2080
- *Scientific Libraries Reference Manual*, publication SR–2081
- *Remote Procedure Call (RPC) Reference Manual*, publication SR–2089
- *Math Library Reference Manual*, publication SR–2138
- *Application Programmer's Library Reference Manual*, publication SR–2165
- *Compiler Information File (CIF) Reference Manual*, publication SR–2401
- *Kerberos User's Guide*, publication SG–2409

General documentation:

- *UNICOS CDBX Symbolic Debugger Reference Manual*, publication SR–2091
- *UNICOS Macros and Opdefs Reference Manual*, publication SR–2403

All publications referenced in this manual are Cray Research publications unless otherwise noted.

The *User Publications Catalog*, publication CP–0099, lists all Cray Research hardware and software manuals that are available to customers.

To order a manual, either call the Distribution Center in Mendota Heights, Minnesota, at (612) 683–5907 or send a facsimile of your request to fax number (612) 452–0141. Cray Research employees may choose to send electronic mail to `order.desk` (UNIX system users) or `order desk` (HPDesk users).

Conventions

The following conventions are used throughout this manual:

Convention	Meaning
`UNDERSCORED UPPERCASE`	Underscored uppercase words in command lines indicate default values.
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
`routine`()	Routine names followed by an empty set of parentheses designate a library or kernel routine; for example, `ddcntl`(). Kernel routines do not have man pages associated with them.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command line.
...	Ellipses indicate that a preceding command-line element can be repeated.
`A|B`	A vertical bar in a command format separates two or more possible choices, one of which you may specify.

Convention	Meaning
Case	UNICOS commands and file names can be either in uppercase or lowercase. The UNICOS operating system distinguishes between the two: `MYFILE` and `myfile` are two different files.
Global	Global SEGLDR directives affect the entire program.
Segment	Segment SEGLDR directives affect only the segment in which they are located.
`O’`	These characters indicate an octal number. For example, `O’177777` means 177777 octal.
Command-line equivalents	These are found at the end of some subsections that describe individual SEGLDR directives. Command-line equivalents are the UNICOS `segldr` command-line options that perform the same function as the SEGLDR directive being described.

The following machine naming conventions may be used throughout this manual:

Term	Definition
Cray PVP systems	All configurations of Cray parallel vector processing (PVP) systems, including the following: CRAY C90 series (CRAY C916, CRAY C92A, CRAY C94, CRAY C94A, and CRAY C98 systems) CRAY C90D series (CRAY C92AD, CRAY C94D, and CRAY C98D systems) CRAY EL series (CRAY Y-MP EL, CRAY EL92, CRAY EL94, and CRAY EL98 systems) CRAY J90 series (CRAY J916 and CRAY J932 systems) CRAY T90 series (CRAY T94, CRAY T916, and CRAY T932 systems)

Term	Definition
	CRAY Y-MP E series (CRAY Y-MP 2E, CRAY Y-MP 4E, CRAY Y-MP 8E, and CRAY Y-MP 8I systems) CRAY Y-MP M90 series (CRAY Y-MP M92, CRAY Y-MP M94, and CRAY Y-MP M98 systems)
Cray MPP systems	All configurations of Cray massively parallel processing (MPP) systems, including the CRAY T3D series (CRAY T3D MC, CRAY T3D MCA, and CRAY T3D SC systems)
All Cray Research systems	All configurations of Cray PVP and Cray MPP systems that support this release
SPARC systems	All SPARC platforms that run the Solaris operating system version 2.3 or later

The default shell in the UNICOS 9.0 release, referred to in Cray Research documentation as the standard shell, is a version of the Korn shell that conforms to the following standards:

- Institute of Electrical and Electronics Engineers (IEEE) Portable Operating System Interface (POSIX) Standard 1003.2–1992
- X/Open Company Standard XPG4

The UNICOS 9.0 operating system also supports the optional use of the C shell.

The POSIX standard uses *utilities* to refer to executable programs that Cray Research documentation usually refers to as *commands*. Both terms appear in this document.

In this publication, *Cray Research*, *Cray*, and *CRI* refer to Cray Research, Inc. and/or its products.

Online information

The following types of online information products are available to Cray Research customers:

- CrayDoc online documentation reader, which lets you see the text and graphics of a manual online. The CrayDoc reader is available on workstations. To start the CrayDoc reader at your workstation, use the `cdoc`(1) command.
- Docview text-viewer system, which lets you see the text of a manual online. The Docview system is available on the Cray Research mainframe. To start the Docview system, use the `docview`(1) command.
- Man pages, which describe a particular element of the UNICOS operating system or a compatible product. To see a detailed description of a particular command or routine, use the `man`(1) command.
- UNICOS message system, which provides explanations of error messages. To see an explanation of a message, use the `explain`(1) command.
- Cray Research online glossary, which explains the terms used in a manual. To get a definition, use the `define`(1) command.
- `xhelp` help facility. This online help system is available within tools such as the Program Browser (`xbrowse`) and the MPP Apprentice tool.

For detailed information on these topics, see the *User's Guide to Online Information*, publication SG–2143.

Reader comments

If you have comments about the technical accuracy, content, or organization of this manual, please tell us. You can contact us in any of the following ways:

- Send us electronic mail from a UNICOS or UNIX system, using the following UUCP address:

  `uunet!cray!publications`

- Send us electronic mail from any system connected to Internet, using the following Internet addresses:

  `pubs0066@timbuk.cray.com` (comments on this manual)

  `publications@timbuk.cray.com` (general comments)

- Contact your Cray Research representative and ask that a Software Problem Report (SPR) be filed. Use `PUBLICATIONS` for the group name, `PUBS` for the command, and `NO-LICENSE` for the release name.

- Call our Software Publications Group in Eagan, Minnesota, through the Technical Support Center, using either of the following numbers:

  1–800–950–2729 (toll free from the United States and Canada)

  +1–612–683–5600

- Send a facsimile of your comments to the attention of “Software Publications Group” in Eagan, Minnesota, at fax number +1–612–683–5599.

- Use the postage-paid Reader’s Comment Form at the back of the printed manual.

We value your comments and will respond to them promptly.

Contents

Page

Preface iii

Related publications iii
Conventions v
Online information viii
Reader comments ix

Introduction [1] 1

Invoking SEGLDR [2] 3

segldr(1) command line 4
ld(1) command line 10
UNICOS environment variable processing 12
  LDDIR variable 12
  LPP variable 13
  MSG_FORMAT variable 13
  NLSPATH variable 13
  SEGDIR variable 13
  TARGET variable 14
  TMPDIR variable 14
Directives processing order 14
Command options and loader directives 15
Differences between segldr and ld 17
Default directives files 19

General Directives [3] 21

Including directives files 22
  INCLUDE directive 23
  LINCLUDE directive 23
Including object modules 24
  BIN directive 26

Page

  LBIN directive 26
  LIB directive 27
  LLIB directive 28
  NODEFLIB directive 28
  DEFLIB directive 29
  FORCE directive 30
  MODULES and SMODULES directives 30
  COMMONS and SCOMMONS directives 31
  DUPORDER directive 32
  OMIT directive 34
The executable program 34
  ABS directive 35
  TRIAL directive 35
Load map control 35
  ECHO directive 36
  Comments 36
  MAP directive 37
  TITLE directive 38
Controlling error messages 39
  MLEVEL directive 39
  USX directive 40
  REDEF directive 40
  DUPENTRY directive 41
  DUPLOAD directive 42
  NODUPMSG directive 42
  NOUSXMSG directive 43
  MSGLEVEL directive 43
Controlling entry points and execution 44
  XFER directive 44
  EQUIV directive 45
  SET directive 46

Page

UNSAT directive 46

Program alignment and initialization 47

ALIGN directive 47

PRESET directive 48

ORG directive 49

Miscellaneous global directives 50

SYMBOLS directive 50

KEEPSYM and HIDESYM directives 50

CASE directive 51

CPUCHECK directive 51

COMPRESS directive 52

LOGFILE directive 53

LOGUSE directive 53

Introduction to Program Segmentation [4] 55

SEGLDR segment tree concept 55

Loader segment tree design 57

Subroutine calling between segments 59

Using segmentation with multitasked programs 63

Segmentation Directives [5] 65

Segment tree definition directives 65

Segment description directives 66

SEGMENT and ENDSEG directives 66

MODULES and SMODULES directives 67

COMMONS and SCOMMONS directives 68

BIN directive 69

SAVE directive 70

DUP directive 71

Global directives for segmentation 72

SLT directive 72

SAVE directive 72

COPY directive 73

Page

SEGORDER directive 74

Program Duplication and Block Assignment [6] 75

Duplication and block assignment in nonsegmented programs 75

Duplicate module names 75

Duplicate entry-point names 75

Duplicate common blocks 76

Block assignment 76

Duplication in segmented programs 76

Module duplication 76

Entry-point duplication 77

Common block duplication 77

Block assignment in segmented programs 79

FLOAT directive 80

Floating 80

Automatic duplication 80

Example 81

Common block use 83

Data load restrictions 83

Block data routines 84

Referencing data in common blocks 84

Segmented Program Execution [7] 85

$SEGRES 85

Subroutine call overhead 86

Dynamic Memory Management [8] 87

Managing global heap memory 87

HEAP directive 87

STACK directives 88

TSTACK directive 89

	Page
ADDBSS directive	90
DYNAMIC directive	90
Using the heap and dynamic common together	92
Fortran example for acquiring space from the heap	92
Fortran example for using dynamic common	93
Central Memory Allocation by SEGLDR [9]	95
Definitions of terms	96
Executable program organization	96
ORDER directive	97
TEXT,DATA,BSS allocation scheme for memory allocation	98
Shared-text allocation scheme for memory allocation	98
Advantages of shared-text programs	99
Disadvantages of shared-text programs	99
Memory allocation for segmented programs	99
ORDER=SS.TDB	100
Soft Externals [10]	103
Soft external references	103
How to declare soft externals	104
How to link soft externals	105
Using soft externals	106
Testing entry-point references with _loaded	106
Testing entry-point references with flag words	107
How to convert soft references to hard references	107
HARDREF directive	108
How to convert hard references to soft references	108

	Page
SOFTREF directive	108
Configuration Directives [11]	109
Specifying default directory search lists	109
DEFDIR directive	109
LIBDIR directive	111
The executable program	112
OUTFORM directive	112
Controlling entry points and execution	113
START directive	113
CALLXFER directive	113
Miscellaneous global directives	114
SYSTEM directive	114
INCFILE directive	114
ZSYMS directive	115
Zero address directives	115
ZEROCOM directive	116
ZERODATA directive	116
ZEROTEXT directive	117
Managing global heap memory	117
DEFHEAP directive	117
DEFSTACK directive	118
FREEHEAP directive	119
Examples [A]	121
Basic case	121
Tree structure examples	121
Tree structure with expandable common block	122
Segmented load with duplicated modules	125
Comprehensive Fortran program example	126
Fortran source code	126
Loader directives	128
SEGLDR map output	130

Page

Program block maps 131

Program entry-point cross-reference map 134

Program common block reference map 135

Messages [B] 137

Loader-created Tables [C] 141

_infoblk 141

Segmentation tables 144

$SEGRES table 145

Segment Linkage table 146

Segment Description table 147

Glossary 149

Index 155

Figures

Figure 1. Segment tree 56

Figure 2. Valid segment tree (broad) 57

Figure 3. Valid segment tree (deep) 58

Figure 4. Invalid segment tree (multiple root segments) 59

Figure 5. Invalid segment tree (multiple immediate-predecessor segments) 59

Figure 6. Invalid segment tree (call across segment tree) 61

Figure 7. Valid and invalid subroutine references 62

Figure 8. Entry-point duplication example 78

Figure 9. Segment tree with duplicate common blocks 79

Figure 10. Segmentation structure before movable block assignment 82

Page

Figure 11. Soft external usage 104

Figure 12. Example tree structure 123

Figure 13. Tree structure 126

Tables

Table 1. Directives equivalents for segldr command-line options 15

Table 2. Directives equivalents for ld command-line options 16

Table 3. segldr and ld differences 17

Table 4. DUPENTRY keywords for duplicated entry definitions 41

Table 5. Subroutine call overhead 86

Table 6. _infoblk description 142

Table 7. $SEGRES description 145

Table 8. SLT description 146

Table 9. SDT description 148

Introduction [1]

The SEGLDR product is a loader for code produced by the CAL version 2, CFT77, CF90, Pascal, Ada, and C language processors. For segmented programs, the SEGLDR product loads program segments as required, without explicit calls to an overlay manager. See “Introduction to Program Segmentation,” page 55, for more information on program segments.

In this manual, *segmented* programs are programs having portions of their code not continuously memory-resident, and *nonsegmented* programs are those having all of their code continuously memory-resident.

Executing under the control of the UNICOS operating system on a Cray Research computer system, the SEGLDR loader produces both segmented and nonsegmented executable programs. Despite its name, the SEGLDR loader is an efficient and full-featured loader for loads that do not require segmentation.

With the SEGLDR loader, you can produce and execute segmented programs without modifying your code extensively. The SEGLDR loader detects subroutine calls that require the loading of new segments into memory. A memory-resident routine, provided by the system and loaded with the object module, manages memory overlays.

“Invoking SEGLDR,” page 3, describes the UNICOS invocation statement.

Invoking SEGLDR [2]

You can control the loader's operation with the invocation statement shown in this section, or with the directives explained in "General Directives," page 21. "Command options and loader directives," page 15, shows the correspondence between command-line options and the loader directives. "Directives processing order," page 14, describes the effects of using both command-line options and directives.

There are two ways to invoke the loader. The `segldr`(1) command provides a simple invocation method in which the loader handles many of the requirements of loading your program. The `ld`(1) command provides a traditional UNIX interface in which you must provide more information to the loader to load your program correctly. The `cc`(1) command uses the `ld` interface when invoking the loader, and the `cf77`(1) command uses the `segldr` interface. "Differences between `segldr` and `ld`," page 17, describes how the two invocation formats differ.

Generally, text in this reference manual refers to "`segldr`" whenever information pertains only to the `segldr` invocation. It uses "`ld`" whenever information pertains only to the `ld` invocation. "SEGLDR" or "the loader" refers to information pertaining to the loader in general.

segldr(1) command line

2.1

Execute the loader with the following command line. Options can be specified in any order, however the order can affect how the options are interpreted:

```
segldr [–A incfile] [–a] [–b value] [–e ename] [–f value] [–g] [–H hi[+he]]
  [–i dirfiles] [–j names] [–k] [–l names] [–m] [–n] [–o outfile] [–s] [–t]
  [–u unames] [–D dirstring] [–E] [–F] [–L ldirs] [–M arguments] [–N]
  [–O keyword] [–S si[+se]] [–V] [–Z] [–z] [objfiles] files
```

–A *incfile* Specifies an existing executable file. segldr extracts the symbols from *incfile* and links the new object modules as a code fragment that will execute as part of the original program.

–a Aligns all local code and data blocks on instruction buffer boundaries.

–b *value* Adds *value* number of 1024-word blocks of memory at the end of the loaded program (BSS space).

–e *ename* Indicates that program execution should begin at entry *ename*. Control is passed from the system startup routine to the *ename* entry point which, under most circumstances, is the user main routine. The –e option is equivalent to the XFER directive.

–f *value* Fills all uninitialized words of the program with *value*, which may be one of the following:

zeros Sets uninitialized data words to 0 (default).

ones Sets uninitialized data words to –1.

indef Sets uninitialized data words to O’0605054000000000000000, which causes a floating-point error if used in a floating-point operation.

–indef Sets uninitialized data words to O’1605054000000000000000, which causes a floating-point error if used in a floating-point operation.

`indefa` Sets uninitialized data to the sum of a logical OR operation of `O'0605054000000000000000` and the address of the word being preset. This value is the same as that of `indef`, except the address of the word referenced appears in the low-order bits of the value.

`–indefa` Sets uninitialized data to the sum of a logical OR operation of `O'1605054000000000000000` and the address of the word being preset. This value is the same as that of `–indef`, except the address of the word referenced appears in the low-order bits of the value.

A 16-bit octal value
Inserts a 16-bit, user-supplied octal value into each parcel of uninitialized data words. The *value* must be in the range `0`<=*value*<=`O'177777`.

`–g` Generates the debug symbol tables and appends them to the executable file. This option is enabled by default. See the `–s` option.

`–H` *hi*[+*he*] Assigns the initial heap values. The *hi* is the initial heap size; *he* is the heap expansion increment.

`–i` *dirfiles* Reads and processes the directives in each of the specified directive files. Separate file specifications with commas. If the file specification begins with a period (`.`) or a slash (`/`), the loader assumes that it is a complete path and uses it without modification. If a dash (–) is present as one of the file names, the loader reads the `stdin` file for directives; otherwise, the loader looks for the named file in the current directory.

–j *names* Reads and processes the directives in each of the specified directive files. Separate file specifications with commas. If the file specification begins with a period (.) or a slash (/), the loader assumes that it is a complete path and uses it without modification. Otherwise, the loader looks for the named file(s) in the segdirs subdirectory in each search path. See the –L option for the list of search directories.

–k Redirects all but summary-class error messages to the load map file. See the –M option.

–l *names* Lists library names. If a name begins with a period (.) or a slash (/), segldr assumes it is a complete path name and uses it without modification as the name of a library file. Otherwise, segldr checks for file lib*name*.a in the list of search directories and includes the first one found as a library file. The list is separated by commas. See the –L option for the list of search directories.

–m Generates an address-level load map and writes it to stdout. Equivalent to –M ,address.

–n Generates a shared text program on Cray PVP systems.

–o *outfile* Writes the executable program to *outfile*. If the –o option is not used, the executable program is written to the file specified by the ABS directive. If neither the –o option nor ABS is specified, the executable output is written to file a.out.

–s Inhibits the generation of debug symbol tables. Debug symbol tables are generated by default.

–t Executes in trial mode. Scans all object modules, checks errors, and generates load maps, but it does not produce an executable program.

`–u` *unames* Enters *unames* as undefined symbols. This is useful for loading from a library, since undefined symbols are needed to force loading of the desired routines.

`–D` *dirstrng* Specifies a character string composed of `segldr` directives. Any global `segldr` directive may be provided. Directives must be separated with semicolons. See “Directives processing order,” page 14, for the order in which `segldr` processes directives.

`–E` Echoes to the load map file all directives processed. See the `–M` option.

`–F` Enables force mode. All modules from `bin` and object files are loaded, whether or not they are referenced.

`–L` *ldirs* Adds one or more directory names to the list of search directories. `segldr` uses the list of search directories to locate files specified with the `–l` and `–j` options, as well as the `LBIN`, `LLIB`, `LINCLUDE`, and `DEFLIB` directives. If the file cannot be located in any specified search directory, `segldr` looks in the directories specified in the default directory search list. See “`DEFDIR` directive,” page 109, for more information on default directory search lists. You may specify up to 100 directory names.

`–M` *file*
`–M ,`*opts*
`–M` *file*`,`*opts* Selects optional load map file and type of map to produce. If a file name is present, the loader writes the load maps to that file in paginated 132-column format. If a file name is not provided, load maps are written to the `stdout` file in nonpaginated, 80-column format. If no load-map options are specified, a block map that is sorted by address is the default type. Load-map options (*opts*) are as follows (you may specify more than one, separated by commas):

`s` or `stat` Lists only load statistics.

`a` or `address` Sorts block map by address; the default map, if no *opt* is specified.

`al` or `alpha` Sorts block map by name.

`b` or `brief` Restricts block map to modules only from object files.

`c` or `cbxrf` Lists common block cross-references.

`e` or `epxrf` Lists entry-point cross-references.

`p` or `part` Lists a combination of `address` and `alpha`.

`f` or `full` Lists all load maps.

`–N` Inhibits inclusion of the default libraries in the load.

`–O` *keyword* Selects the memory allocation order, which may be as follows:

`tdb` Allocates all code, followed by all initialized data, followed by all uninitialized data (text, data, BSS).

`ema` Allocates code to maximize use of Cray PVP systems extended memory addressing.

`s` Allocates code to create a shared-text program for Cray PVP systems.

`ss.ema` Allocates code to create a split-segment program that maximizes use of Cray PVP systems extended memory addressing.

`ss.tdb` Allocates code to create a split-segment program, where code is followed by initialized data, which is followed by uninitialized data (text, data, BSS).

`–S` *si*[+*se*] Assigns initial stack values. The *si* is the initial stack size; *se* is the initial stack expansion increment.

`–V` Indicates that the loader list its version line to the `stderr` file.

`–Z` Inhibits `segldr` from reading the default directives file `/lib/segdirs/def_seg`. The default directives file is required to configure programs correctly for execution under the UNICOS operating system. The `–Z` option should only be used by special-purpose programs.

`–z` Specifies an alternative default directives file. The alternative directives must configure the program correctly for execution under the UNICOS operating system.

objfiles Files containing object modules produced by the compilers or assembler and object module library files prepared by `ar`(1) or `bld`(1) can be specified. Specifying files on the command line has the same effect as specifying them in a `BIN` directive.

files Files to be loaded. They may contain any of the following:

- Sequential object modules produced by the compilers or assembler. Specifying an object file on the command line has the same effect as specifying it on a `BIN` directive.
- Object libraries produced by `ar`(1) or `bld`(1). Specifying a library on the command line has the same effect as naming it on a `BIN` directive.
- SEGLDR directives. If you enter a hyphen (-) instead of file names, SEGLDR will accept directives from `stdin`.

Load maps, if selected, are written to the `stdout` file by default (see the `–M` option). Error messages are written to the `stderr` file by default (see the `–k` option).

Any file named in the loader directives or command line may be described by a full file path name.

`ld`(1) command line
2.2

To invoke the loader with a command-line format and defaults similar to those of the traditional UNIX `ld`(1) command, you can use the `ld`(1) command.

You can specify options in any order, however the order may affect how the options are interpreted (see `–l` and `–L`). Options and file arguments may be intermixed on the command line.

```
ld [–D dirstring] [-e name] [–F] [–g] [–i] [–j names][–l names] [–L ldirs]
  [–m] [–n] [–o outfile] [-r] [-s] [–u unames] [–V] [–Z] [–z file] files
```

`–D` *dirstring* Specifies a character string composed of the loader directives separated with semicolons. See “Directives processing order,” page 14, for the order in which the loader processes directives.

`–e` *name* Sets the program entry address to the value of symbol *name*. Control is passed from the system kernel to the *name* entry point which, under most circumstances, is the system startup routine. The `–e` option is equivalent to the `START` directive.

`–F` Enables default library processing. The standard system libraries are processed after any user-supplied libraries. Processing of the system libraries is disabled by default.

`–g` Generates the debug symbol tables and appends them to the executable program. This option is enabled by default. See the `–s` option.

`–i` Generates a shared-text program on Cray PVP systems. Equivalent to the `–n` option.

`–j` *names* Lists directives file names. The list is separated by commas. If a name begins with a period (`.`) or a slash (`/`), `ld` assumes it is a complete path name and uses it without modification. Otherwise, `ld` checks for a `segdirs`/*name* file in the list of search directories and uses the first one found. See the `–L` option for the list of search directories.

`–l` *names* Lists library names. If a name begins with a period (`.`) or a slash (`/`), `ld` assumes it is a complete path name and uses it without modification as the name of a library file; otherwise, `ld` checks for file *libname*`.a` in the list of search directories and includes the first one found as a library file. The list is separated by commas. See the `–L` option for the list of search directories.

`–L` *ldirs* Adds one or more directory names to the list of search directories. `ld` uses the list of search directories to locate files specified with the `–l` and `–j` options, as well as the `LBIN`, `LLIB`, `LINCLUDE`, and `DEFLIB` directives. If the file cannot be located in any specified search directory, `ld` looks first in `/lib` and then in `/usr/lib`. You may specify up to 100 directory names.

`–m` Generates a load map of the executable program and writes it to the `stdout` file.

`–n` Generates a shared-text program on Cray PVP systems. Equivalent to the `–i` option.

`–o` *outfile* Writes the executable program to *outfile*. The default *outfile* name is `a.out`.

`-r` Produces relocatable output from prior linked `.o` files. The output is suitable for use by another invocation of `ld`. It is equivalent to using the following directives:

```
OUTFORM=REL
USX=NOTE
SYSTEM=STDALONE
ZSYMS=OFF
```

`–s` Inhibits generation of debug symbol tables. Debug symbol tables are generated by default.

`–u` *unames* Enters *unames* as undefined symbols. This is useful for loading from a library, because undefined symbols are needed to force loading of the desired routines.

–V Lists the ld(1) version line on stderr.

–Z Inhibits ld from reading the default directives file /lib/segdirs/def_ld. The default directives file is required to configure programs correctly for execution under the UNICOS operating system. The –Z option should be used only by special-purpose programs.

–z *file* Specifies an alternate default directives file. The alternate directives must configure the program correctly for execution under the UNICOS operating system.

files Files to be loaded. They may contain any of the following items:

- Sequential object modules produced by the compilers or assembler. Specifying an object file on the command line has the same effect as specifying it on a BIN directive.
- Object libraries produced by ar(1) or bld(1). Specifying a library on the command line has the same effect as naming it on a LIB directive.
- ld directives

UNICOS environment variable processing 2.3

Seven environment variables affect the execution of the loader: LDDIR, LPP, MSG_FORMAT, NLSPATH, SEGDIR, TARGET, and TMPDIR.

LDDIR *variable* 2.3.1

The LDDIR variable lets you specify ld directives or files of directives that are included automatically each time that you use ld. Thus you can set up your own defaults, tailored to the way you use ld. LDDIR is recognized only when ld is invoked.

Set the LDDIR variable by using the following format:

string;*string*;*string*;...

Each *string* is either a ld directive or the name of a file containing ld directives. See “Directives processing order,” page 14, for a discussion of the order in which directives are processed.

LPP *variable*
2.3.2

If LPP is defined, the loader uses the value of the variable as the number of lines to print on each page for listing output. The LPP value must be between 15 and 999. If LPP is not present, the default is 57 lines per page.

MSG_FORMAT *variable*
2.3.3

The MSG_FORMAT variable describes a printing format similar to the C library routine, printf, that can be used to alter the layout of error messages produced by the loader. See the explain(1) command for a complete description of MSG_FORMAT.

NLSPATH *variable*
2.3.4

The NLSPATH variable specifies a list of alternative directories that the loader should search to locate its error message catalog. The NLSPATH environment variable is used to select alternative catalogs for debugging purposes, or when different versions of the loader are operating on the same system. It is not needed for normal operation.

SEGDIR *variable*
2.3.5

The SEGDIR variable lets you specify segldr directives or files of directives that are included automatically each time that you use segldr. Thus you can set up your own defaults, tailored to the way you use segldr. SEGDIR is recognized only when segldr is invoked.

Set the SEGDIR variable by using the following format:

string;*string*;*string*;...

Each *string* is either a segldr directive or the name of a file containing segldr directives. See “Directives processing order,” page 14, for a discussion of the order in which directives are processed.

TARGET *variable*
2.3.6

The TARGET variable specifies the machine characteristics of the system on which the program will execute. The loader generates the program so that it operates correctly on that system. If the TARGET variable has not been specified, the program is adapted to the host system. See target(1) for more information.

TMPDIR *variable*
2.3.7

The TMPDIR variable specifies the directory that the loader uses for its temporary file. If the variable is not specified or is not correct, a site-specific system default is used.

Directives processing order
2.4

The segldr and ld invocations of the loader process directives and command-line options in a similar manner. This subsection describes the order of processing and how directives interact with command-line options. Directives and command-line options are processed in the following order:

1. The loader first reads and processes the default directives file, which provides the loader with the basic information needed to construct a valid UNICOS executable program. The contents of the file may be tailored to meet the needs of each site. The –Z command-line option can be used to inhibit default processing of this file. The –z command-line option may be used to provide an alternative default directives file. The default directives files are:

   ```
   segldr     /lib/segdirs/def_seg
   ld         /lib/segdirs/def_ld
   ```

2. After the default directives file is processed, segldr interrogates the SEGDIR environment variable; ld interrogates the LDDIR environment variable. Directives and directives file names may be specified in the environment variable. The directives and file contents are processed in the order encountered.

3. The command line is processed next. Each command-line option has an equivalent directive that performs the same function. Table 1 describes the correspondence between segldr command-line options and directives. Table 2, page 16, provides the correspondence between ld command-line options and directives. Command-line options and

arguments are processed in the order encountered, with one exception: directives files specified on the command line, either as arguments or with the `segldr –i` option, are processed after all other command-line options.

Because segmentation directives must be evaluated after global directives, they can be specified only in the user directives files named on the command line. User directives files can be specified either as command-line arguments or with the `–i` command-line option.

Command options and loader directives
2.5

Table 1 and Table 2, page 16, show the correspondence between `segldr` and `ld` command-line options and loader directives.

Table 1. Directives equivalents for `segldr` command-line options

Command-line option	Directive
`a`	`align=modules`
`b` *value*	`addbss=`*value*
`e` *entry*	`xfer=`*entry*
`f` *value*	`preset=`*value*
`g`	`symbols=on`
`i` *file*	`include=`*file*
`j` *name*	`linclude=segdirs/`*name*
`k`	no directive equivalent
`l` *name*	`lib=`*libname*`.a`
`l` `/`*filename*	`lib=/`*filename*
`m`	`map=address`
`n`	`order=shared`
`o` *file*	`abs=`*file*
`s`	`symbols=off`
`t`	`trial`
`u` *name*	`unsat=`*name*

Table 1. Directives equivalents for `segldr` command-line options
(continued)

Command-line option	Directive
`z`	no directive equivalent
`A` *file*	`incfile=`*file*
`D` *directive*	*directive*
`E`	`echo=on`
`F`	`force=on`
`H` *values*	`heap=`*values*
`L` *directory*	`libdir=`*directory*
`M` `,`*keywords*	`map=`*keywords*
`N`	`nodeflib`
`O` *keyword*	`order=`*keyword*
`S` *values*	`stack=`*values*
`V`	No directive equivalent
`Z`	No directive equivalent
`.o` object file argument	`bin=`*file*
`.a` library file argument	`bin=`*file*
Directives file argument	`include=`*file*

Table 2. Directives equivalents for `ld` command-line options

Command-line option	Directive
`e` *entry*	`start=`*entry*
`g`	`symbols=on`
`i`	`order=shared`
`j` *name*	`linclude=segdirs/`*name*
`l` *name*	`llib=`*libname*`.a`
`l` `/`*filename*	`lib=/`*filename*
`m`	`map=address`

Table 2. Directives equivalents for ld command-line options
(continued)

Command-line option	Directive
n	order=shared
o *file*	abs=*file*

r	outform=rel;usx=note;
	system=stdalone

s	symbols=off
u *name*	unsat=*name*
z	No directive equivalent
D *directive*	*directive*
F	include=ld_Flib
L *directory*	libdir=*directory*
V	No directive equivalent
Z	No directive equivalent
.o object file argument	bin=*file*
.a library file argument	lib=*file*
Directives file argument	include=*file*

Differences between segldr and ld 2.6

In addition to differences in command-line invocation formats, segldr and ld vary in other ways. Table 3 summarizes these differences.

Table 3. segldr and ld differences

Feature	segldr	ld
Default directives file	/lib/segdirs/def_seg	/lib/segdirs/def_ld
Environment variable processing	SEGDIR	LDDIR

Table 3. segldr and ld differences
(continued)

Feature	segldr	ld
Object file processing	All object file names are included as bin files.	All .o files (sequential object files) are included as bin files. All .a files (library object files) are included as lib files.
Default setting of DUPENTRY directive	DUPENTRY=CAUTION:CAUTION: NOTE	DUPENTRY=CAUTION:NOTE:NOTE. Because of the different dupentry setting, and the practice of including library object files as lib files, ld issues fewer diagnostic messages about duplicated entry point names than segldr.
Default setting of DUPORDER directive	DUPORDER=OFF The first definition of an entry point is chosen, regardless of the definition's location.	DUPORDER=ON. An ordered search algorithm is used. The entry point that ld chooses depends on the order of definitions and references. See "DUPORDER directive," page 32, for more information.
Default system libraries	A list of default libraries is included. Most common system routines are included in these libraries.	No default libraries are included. You must specify all libraries required by your program.
Default setting for USX directive	USX=CAUTION. A program that contains unsatisfied external references is still executable and segldr exits normally. Calls to unsatisfied references are intercepted when the program is run.	USX=WARNING. A program that contains unsatisfied external references is not executable and ld exits with a nonzero error status.
Default setting for FORCE directive	FORCE=OFF. Modules in bin files are included in the executable program only if they are referenced, contain a main program, or initialize global data.	FORCE=ON. All modules encountered in bin files are included in the executable program, whether or not the modules are referenced.

Default directives files
2.7

`segldr` and `ld` begin processing by reading a file of directives. The `segldr` default directives file is `/lib/segdirs/def_seg`; the `ld` default directives file is `/lib/segdirs/def_ld`. The defaults directives files provide the basic information needed for `segldr` or `ld` to create an executable UNICOS program. In addition, directives can be added to the default files to meet the loader operations needs of a particular site. Several common options for modifying the default directives files include the following:

- Adding or deleting default libraries
- Adding or deleting search directives
- Changing message severities

Defaults for directives are discussed throughout this manual. These settings reflect the values as released by Cray Research. The default values you find at your site may differ.

You can suppress default directives file processing by including the `–Z` option on your `segldr` or `ld` command line. You can substitute a different directives file by using the `–z` option. If you choose to substitute the directives file, you must provide the necessary directives to cause the loader to correctly build your program.

General Directives [3]

The loader directives identify relocatable object files to be loaded, select various control options, and declare the segmentation structure. When using the `segldr` command, you can specify files of `segldr` directives with the `–i` option or you can specify directives themselves with the `–D` option.

The loader recognizes the following groups of directives, which should be specified in the indicated order:

1. Global directives identify relocatable object files to be loaded and select various options that control the load process. Most of the global directives are described in this section; global and segment directives are also discussed in sections 5, 8, 9, 10, 11, and 12. Global directives can be entered in any order, but all global directives must precede all other directives.

2. Segment tree definition directives should follow the global directives and are described in, “Segment tree definition directives,” page 65.

3. Segmentation directives specify the structure of segmented programs, should follow tree-definition directives, and are described in, “Segment description directives,” page 66.

Most loader directives have *KEYWORD=value* syntax. Exceptions are stated in individual directive descriptions. The following describes the conventions used in representing loader directives:

- You can enter directives and keywords in uppercase or lowercase, but not in mixed case. Files, modules, entry points, and common blocks can be specified in uppercase, lowercase, or mixed case; however, under the UNICOS operating system, the loader treats file names and module names of different cases as different names. You can use the `CASE` directive to change the way in which the loader interprets lowercase directives.

- Comments can appear anywhere in the input directives. Each comment must be preceded with an asterisk (`*`), and all characters to the right of the asterisk are not processed.

- Terminate directives with a semicolon (;), an asterisk (*), or an end-of-line character.
- More than one directive can appear on a single line, but you must separate multiple directives on a single line with a semicolon.
- A directive cannot be longer than 256 characters.
- Separate elements in a list with commas.
- The loader ignores null directives (for example, two successive semicolons or a blank line).
- Some loader directives can consist of more than one line. These directives have a comma as the last nonblank character before the end-of-line character. See individual directive descriptions for more detail.
- The loader normally uses such special characters as semicolons (;), commas (,), and others as delimiting characters when processing directives. If you want to use any of these characters (except semicolons) in the names of files, entry points, common blocks, or modules, place the complete name within single or double quotation marks. For example: `bin='`*abc*`:`*def*`.o'`
- Because semicolons are used to separate directives, they cannot be included in literal strings (strings enclosed in quotation marks).

Including directives files
3.1

The `INCLUDE` and `LINCLUDE` directives allow you to specify the names of files that contain directives for the loader to process. When an `INCLUDE` or `LINCLUDE` directive is encountered, the loader stops reading the current directives files and begins reading the file specified with the `INCLUDE` or `LINCLUDE` directive. When the end of the included directives file is reached, the loader resumes processing the original file, using the directive that follows the `INCLUDE` or `LINCLUDE` directive. `INCLUDE` or `LINCLUDE` directives can appear in included files, up to a maximum of 10 nesting levels.

INCLUDE *directive*
3.1.1

The INCLUDE directive specifies a file that should be included in the load process.

Format:

```
INCLUDE=file
```

Example:

```
MAP=stat
INCLUDE=dirfile1
DUPLOAD=caution
```

In this example, the loader processes the MAP=stat directive, then it processes the directives found in dirfile1, and lastly it processes the DUPLOAD=caution directive.

LINCLUDE *directive*
3.1.2

The LINCLUDE directive specifies a file that should be included in the load process. Only the file name component should be specified. The loader scans the list of search directories to locate the file. (See "LIBDIR directive," page 111, for information on user directory search lists.)

Format:

```
LINCLUDE=file
```

Example:

```
LIBDIR=/mydir/lib
LINCLUDE=dirfile2
```

In this example, the loader searches for file /mydir/lib/dirfile2. If it is found, the directives in dirfile2 is processed. Otherwise, the loader looks for /lib/dirfile2, then /usr/lib/dirfile2. It uses the first of these files it finds.

Including object modules
3.2

The BIN, LBIN, LIB, and LLIB directives let you identify the relocatable modules that you want the loader to include in your program. The DUPORDER directive lets you determine how to select the modules to be retrieved from libraries. The NODEFLIB and OMIT directives provide control over the system default libraries. The FORCE, MODULES, and COMMONS directives provide you with additional control over the loading process.

Files specified in BIN or LBIN directives or specified as command-line arguments by the loader are all considered to be bin files. Segmented object files specified as arguments on the ld command line are also considered to be bin files. By convention, bin files should be the portion of your program that you have written. Files specified in LIB, LLIB or DEFLIB directives, or specified with the –l option on the segldr or ld command line, are all lib files. Library files built by bld and specified as arguments on the ld command line are also considered lib files. By convention, lib files are libraries of previously written routines that the loader includes in your program as needed. The loader processes bin and lib files in a very similar manner: it scans all modules from both bin and lib files, and it establishes and retains the calling relationships between all modules. After processing all files in this way, the loader determines which modules must be loaded. It begins at the module containing the transfer entry address and scans the calling relationships, retaining all modules that are called and deleting all others. Exceptions and differences between bin and lib file processing are as follows:

- All bin files are processed before all lib files. If modules containing duplicate entry points are discovered, the loader uses the first occurrence. See “DUPORDER directive,” page 32.
- The FORCE directive causes the loader to include all modules from bin files, even if they are not referenced. FORCE does not affect modules from lib files. See “FORCE directive,” page 30.
- The BRIEF option to the MAP directive limits load maps to modules derived from bin files. Modules from lib files are not listed. See “MAP directive,” page 37.
- The DUPORDER directive affects the selection of modules from library files. See “DUPORDER directive,” page 32.
- The DUPENTRY directive controls messages concerning duplicate definitions of the same entry point. It differentiates between entry points from bin files and those from lib files. See “DUPENTRY directive,” page 41.

- Fortran BLOCKDATA subprograms encountered in bin files are always included in the program. BLOCKDATA subprograms encountered in lib files are included only if they are referenced.
- The loader always includes a module written in C and encountered in a bin file if the module initializes global data. C modules from lib files that initialize global data are included only if they are referenced.

In addition to the files you provide, the loader also scans a set of default system libraries. You can use the NODEFLIB directive to inhibit default library processing.

The default libraries that segldr and ld scan and the order of scanning are specified in the default directives files. The default directives files as released by Cray Research specify processing the libraries in the order listed:

```
libc.a
libu.a
libm.a
libf.a
libfi.a
libsci.a
libp.a
```

Some of the default libraries listed above may be released separately from the UNICOS operating system; therefore, they may not be present on your system. Missing libraries are silently ignored.

The loader uses a directory search algorithm to locate each default lib file. If you have provided a list of search directory names by using the –L option or LIBDIR directive, the loader searches the directories specified to locate the default libraries. If the libraries cannot be located in those directories, the loader searches the directories specified in the default directory search list. (See “DEFDIR directive,” page 109, for information on the default directory search list.)

BIN *directive* 3.2.1

The BIN directive names the relocatable object input files to be searched. Multiple BIN directives have a cumulative effect. If you specify multiple files with BIN, the loader processes them in the order specified.

If a module is present in more than one file, the loader loads the first module encountered. However, if you use the MODULES directive and specify a particular file, this rule may not apply.

Format:

BIN=$file_1$[,$file_2$,$file_3$,...,$file_n$]

$file_i$ Names of relocatable input files to be included. If no bin files are specified, the default is a.o.

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

Examples:

```
bin=myfile,../group/ourfile.o,
     ../sue/herfile.a,/u/steve/anyfile.o

bin=newfile.a,oldfile.a
```

Modules contained in global BIN files (as opposed to segmented BIN files) are not assumed to be in any particular segment, unless the module is specified in a segmented MODULES directive.

Command-line equivalent: *objfiles* argument

LBIN *directive* 3.2.2

The LBIN directive, in a manner similar to the BIN directive, names relocatable object input files for the loader to search. With LBIN, however, only the file name component is specified. The LIBDIR directory search applies to names on the LBIN directive. Each LIBDIR directory is searched for the files specified. The first file found is included in the program.

Format:

`LBIN=`file_1`[,`file_2`,`file_3`,...,`file_n`]`

file_i Names of the files you provide.

`LIB` *directive* 3.2.3

The `LIB` directive names the relocatable object library files for the loader to search when the loader is trying to find entry points that are referenced in `BIN` files but are not defined in any `BIN` files or previously searched `LIB` files.

Use the `LIB` directive to specify `lib` files in addition to those in the loader's default list of libraries. Library files specified with the `LIB` directive are searched in the order specified and before any default libraries.

The effect of multiple `LIB` directives is cumulative.

If you continue this directive beyond one line, end each continued line with a comma. The `LIBDIR` directory search does not apply to files specified in `LIB` directives. Each name should be a complete path name.

Format:

`LIB=`lib_1`[,`lib_2`,`lib_3`,...,`lib_n`]`

lib_i Names of the libraries you provide.

Examples:

```
lib=/u/lib/lib7.a,/u/lib/libarf.a,
     /lib/lib3A.a,mytmplib.a,mylibY.a

lib=mylibs.o,/lib/libc.a
```

These examples each specify seven libraries that the loader should search before searching the default libraries. The libraries are searched in the order given.

LLIB *directive*

3.2.4

The LLIB directive, in a manner similar to the LIB directive, specifies relocatable object libraries for the loader to search. With LLIB, however, only the file name component is specified. The LIBDIR directory search applies to what is specified on the LLIB directive. Each LIBDIR directory is searched for the files specified. The first file found is included in the program.

Format:

LLIB=*name*[,*name*,...]

Command-line equivalent: -l option

Example:

```
LIBDIR=/lib/xlib
LLIB=libscan.a
```

First, the loader looks for file /lib/xlib/libscan.a, then /lib/libscan.a, and finally /usr/lib/libscan.a. It uses the first of these files it finds.

NODEFLIB *directive*

3.2.5

The segldr default directives file contains a set of DEFLIB directives. NODEFLIB instructs the loader to ignore some or all of the libraries that have been specified by DEFLIB directives. If all default libraries are to be ignored, only modules found in files declared as BIN or LIB files are considered for loading.

Format:

NODEFLIB
  NODEFLIB=*deflib*$_1$[,*deflib*$_2$,...,*deflib*$_n$]

If the first format is used, all default libraries are ignored. If the second format is used, only the specified default libraries are ignored.

> **Note:** For a segmented load, you must specify the library containing the loader run-time routine $SEGRES.

Example:

```
NODEFLIB
LIB=/lib/libio.a,/lib/libc.a
```

The preceding example tells the loader to search the libraries specified by the `LIB` directive (in the order specified) for unsatisfied externals. The loader does not search the default libraries for entry points not found in the specified libraries.

Example:

```
NODEFLIB=libp.a
```

This example directs the loader to ignore the Pascal library, and to process the other default libraries as usual.

Command-line equivalent: `–N` option

`DEFLIB` ***directive*** 3.2.6

The `DEFLIB` directive instructs the loader to add libraries to its list of default libraries. Each library specified in the `DEFLIB` directive is added to the end of the list of default libraries. If `DEFLIB` specifies a library that is already part of the default library list, the loader moves that library name to the end of the list. You may use `NODEFLIB` and `DEFLIB` together to replace some or all of the default system libraries (See “Including object modules,” page 24). All libraries specified by the `DEFLIB` directive are processed after all libraries that are specified by the `LIB` directive are processed.

Format:

`DEFLIB=`$deflib_1$`[,`$deflib_2$`,...,`$deflib_n$`]`

$deflib_i$ Name of one library to add.

Example:

```
DEFLIB=libmylib.a
```

This example directs the loader to add the user’s library to the end of the default library list.

Example:

```
NODEFLIB; DEFLIB=libuser.a
```

This example suppresses all normal default system libraries, replacing them with one user library.

FORCE *directive* 3.2.7

The loader gathers all modules in all files specified with global BIN and LIB directives. It then discards all modules with entry points that are never called. FORCE specifies that subprograms not called by other subprograms are to be loaded anyway (force-loaded). This can be helpful in debugging, letting you force-load a debug routine not actually called by the program.

Format:

```
FORCE=ON|OFF
```

ON Enables force-loading; when FORCE=ON, the loader loads all modules specified in MODULES directives and in all bin files.

OFF Disables force-loading; when FORCE=OFF, the loader discards modules to which no references have been made (except the XFER directive's module and the BLOCKDATA subprograms found in bin files) (default).

Command-line equivalent: –F option

MODULES *and* SMODULES *directives* 3.2.8

The MODULES and SMODULES directives specify modules to load. Normally, if more than one module with a particular name exists, the loader chooses the first such module it encounters. If modules of the same name are encountered in different files, you can use the MODULES directive to specify the files from which the modules are obtained.

Additionally, you can use MODULES to specify the loading order in a nonsegmented load. Loading order can be affected by other considerations such as the current memory ordering algorithm. See “Executable program organization,” page 96. If you use the MODULES directive, an error message will be issued if the modules specified cannot be located in any included file. Error messages are not issued if SMODULES is used.

Format:

MODULES=$modname_1$[:$file_1$][,$modname_2$[:$file_2$],...,$modname_n$[:$file_n$]]
SMODULES=$modname_1$[:$file_1$][,$modname_2$[:$file_2$],...,$modname_n$[:$file_n$]]

$modname_i$ Name of the module to be loaded.

$file_i$ Name of the file from which to obtain the module.

Example:

```
MODULES=SUBA,SUBB:myfile.o,SUBC
MODULES=SUBD:lib1.a
```

In the preceding example, the MODULES directive tells the loader to obtain SUBB from file myfile.o and to obtain SUBD from file lib1.a; modules SUBA and SUBC are obtained from the first file in which each is encountered.

COMMONS *and* SCOMMONS *directives*
3.2.9

In an unsegmented program, COMMONS and SCOMMONS cause the listed common blocks to be loaded in the indicated order. In a segmented load, however, the COMMONS directive serves only to order and/or set the size of common blocks. Loading order can be affected by other considerations such as the current memory ordering algorithm. See “Executable program organization,” page 96.

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

If you use the COMMONS directive, an error message will be issued if the indicated common blocks cannot be located in any included file. No error messages are issued if SCOMMONS is used.

Format:

```
COMMONS=blkname1[:size1][,blkname2[:size2 ],...,blknamen[:sizen]]
SCOMMONS=blkname1[:size1][,blkname2[:size2 ],...,blknamen[:sizen]]
```

blkname$_i$ Name of the common block to be loaded.

size$_i$ Decimal number indicating the size of the common block. If present, it overrides any common block sizes declared in your code. If the size specified is 0, the first common block size encountered in your code (for this common block) is used. By default, the loader uses the longest common block definition it encounters in those modules of your code that are actually referenced and loaded.

DUPORDER *directive*
3.2.10

The DUPORDER directive selects the method the loader uses to process duplicated entry points found in libraries. When processing BIN files, the loader always chooses the first occurrence of a duplicated entry point. If the duplicated symbol appears in both a BIN and a LIB file, the loader always chooses the one in the BIN file. If the duplicated symbol appears only in library files, the loader has two methods of selecting the occurrence of the symbol to use: if the DUPORDER directive is not enabled (OFF, default for `segldr`), the loader uses the first occurrence of the symbol. If the DUPORDER directive is enabled (ON, default for `ld`), the loader uses *ordered duplicate selection*, which means that the loader locates the first module that references the duplicated symbol and then looks for a definition of the symbol in succeeding modules. The first definition found in a succeeding module is the one used. If the loader finds no succeeding definition, the first definition encountered anywhere is used.

Format:

```
DUPORDER=ON|OFF
```

`ON` The loader uses *ordered duplicate selection* to choose the entry point to use (default for `ld`).

`OFF` The loader uses the first occurrence of the duplicated entry point (default for `segldr`).

The following example, in which a partial Fortran program is loaded, contrasts the `ON` and `OFF` settings of the `DUPORDER` directive.

```
module 1:  PROGRAM DUPEXAMP
              .   .   .
           CALL REFMOD
              .   .   .
           END

module 2:  SUBROUTINE DUPLICAT
              .   .   .
           END

module 3:  SUBROUTINE REFMOD
           CALL DUPLICAT
           END

module 4:  SUBROUTINE DUPLICAT
              .   .   .
           END
```

Module 1 contains the main program and is included in the load in a binary file. Modules 2, 3, and 4 occur, in the order shown, in library files. If the `DUPORDER` directive is disabled (`OFF`, or not used), the loader selects the `DUPLICAT` symbol in module 2 to satisfy the reference in module 3, because it is the first occurrence of the symbol. If the `DUPORDER` directive is enabled (`ON`), the loader selects the `DUPLICAT` symbol from module 4 because this is the first definition for `DUPLICAT` that occurs after the reference in module 3.

`OMIT` *directive* 3.2.11

The `OMIT` directive specifies modules that should be bypassed by the loader when processing object or library files.

Format:

```
OMIT=module1[:file1][,module2[:file2],...]
```

Modules specified on the `OMIT` directive are not included in the program, even if referenced from other modules. If *file* is not present, all modules with the indicated name are omitted, regardless of the file in which they are found. If *file* is specified, only *module* from that file is omitted. Modules with the same name, but in different files, are included.

If a module is omitted, and the program makes references to symbols within that module (that cannot be satisfied by any other module), the reference is treated in the same manner as any other unsatisfied reference.

Example:

```
    omit=printf$c:/lib/libc.a,mymodule
```

In this example, the `printf$c` module, from file `/lib/libc.a` and from any module with the name `mymodule`, is bypassed in the load process.

The executable program 3.3

The `ABS` and `TRIAL` directives give you a measure of control over the executable program that the loader produces. You can tell the loader where to write the executable file, or whether the loader should produce the executable file or only a load map and error messages.

ABS *directive*
3.3.1

The ABS directive specifies the file to receive the executable program constructed by the loader.

Format:

```
ABS=file
```

file The *file* parameter specifies the file to receive the executable program. The default is `a.out`.

Command-line equivalent: `–o` option

TRIAL *directive*
3.3.2

The TRIAL directive lets you make a sample of the loader run without creating any executable output. You can therefore print a load map and most error messages without using a lot of memory to build the executable output. Making test runs with TRIAL also lets you determine optimal memory use for data areas or identify total memory requirements for a particular application.

Format:

```
TRIAL
```

Command-line equivalent: `–t` option

Load map control
3.4

The ECHO, COMMENT, MAP, and TITLE directives control the information that the loader writes to the listing output file. The default listing output file is `stdout`. You can change these defaults by using the `–M` option.

`ECHO` *directive* 3.4.1

The `ECHO` directive resumes or suppresses the printing of input directives.

Format:

```
ECHO=ON|OFF
```

`ON` Resumes the listing of input directives.

`OFF` Suppresses directive listing. If `ECHO=OFF`, the loader automatically echoes erroneous directive lines, followed by the error message (default).

Comments 3.4.2

Comments annotate the loader directives. They are echoed to the listing file but are otherwise ignored. All characters to the right of the asterisk are considered part of the comment string.

The asterisk character begins a comment. You can use comments in either the global or the segment description directives, but you cannot embed comments within directives.

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
* Segment Description Directives follow
```

MAP ***directive***
3.4.3

The MAP directive controls the loader map output generation. Besides memory mapping, MAP provides the time and date of load, the length of the longest branch and the last segment, and the transfer address. Map output is written to the listing file. See “Examples,” page 123, for more information on map output.

Format:

MAP=[*keyword*$_1$, ..., *keyword*$_n$]

NONE Writes no map output to the listing file (default).

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 it lists modules by ascending load address.

PART Writes both ALPHA and ADDRESS information.

EPXRF	Writes the STAT information plus the Entry Point Cross-reference table.
CBXRF	Writes the STAT information plus the Common Block Cross-reference table.
FULL	Writes all PART, EPXRF, and CBXRF information.
BRIEF	Limits information in the ADDRESS and ALPHA output to modules from bin files.

The effects of multiple keywords are cumulative.

Command-line equivalents: –m and –M options.

TITLE *directive* 3.4.4

The TITLE directive places an arbitrary, user-defined character string in the second line of each page header. TITLE forces a page eject and then writes the header lines at the top of the new page.

The title line is initially clear, and it can be reset by TITLE directives in either the global or the segment description directives portion of the input. An end-of-line or a semicolon (;) signals the end of the TITLE string.

Format:

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

title string User-defined character string; maximum length is 74 characters. If no *title string* is specified, the title line is cleared.

Example:

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

This TITLE directive copies the string “Place this in the page header, please.” to the page header.

Controlling error messages
3.5

The MLEVEL, USX, REDEF, DUPENTRY, DUPLOAD, NODUPMSG, NOUSXMSG and MSGLEVEL directives let you control the printing of error messages. Error messages are written to stderr by default, although you can redirect them to other files by using standard I/O redirection or with the loader –k command-line option.

MLEVEL *directive*
3.5.1

The MLEVEL directive controls the loader messages on the listing output. The keyword indicates the lowest-priority message to be printed. If you do not use the MLEVEL directive, MLEVEL=CAUTION is assumed.

Format:

MLEVEL=*keyword*

FATAL Prints only FATAL-level messages. When a message with this severity level is issued, the loader is terminated immediately, and no executable output is written.

WARNING Prints FATAL- and WARNING-level messages. A WARNING-level message indicates that the executable output may not be written; if the output is written, it is not executable. Processing continues so that additional messages may be printed.

CAUTION Prints FATAL-, WARNING-, and CAUTION-level messages. A CAUTION-level message indicates that an error may have occurred, but it is not severe enough to prohibit generation of executable output (default).

NOTE Prints FATAL-, WARNING-, CAUTION-, and NOTE-level messages. A NOTE-level message indicates that the loader may have been misused or used inefficiently; it has no effect on execution validity.

COMMENT Prints all levels of messages. A COMMENT-level message does not affect execution.

`USX` *directive*

3.5.2

The `USX` directive lets you determine the severity level of unsatisfied external symbols. `CAUTION` is the default.

Format:

`USX=`*keyword*

`FATAL`, `WARNING`, `CAUTION`, `NOTE`, `COMMENT`

See the descriptions for these in “`MLEVEL` directive,” page 39.

`IGNORE` This is the same as `COMMENT`.

`REDEF` *directive*

3.5.3

The loader generates an error message if you redefine common blocks with varying lengths in different modules.

`REDEF` lets you control the severity level of the loader’s messages when common blocks are defined with varying sizes. The loader always takes the longest definition, regardless of the `REDEF` value. The severity level you select applies to cases in which the common block is redefined with a larger size. The severity level is one level lower for cases in which the common block is redefined with a smaller size.

Format:

`REDEF=`*keyword*

`FATAL`, `WARNING`, `CAUTION`, `NOTE`, `COMMENT`

See the descriptions for these in “`MLEVEL` directive,” page 39.

`IGNORE` This is the same as `COMMENT`.

DUPENTRY *directive*
3.5.4

The DUPENTRY directive controls the severity of the message generated when the loader encounters a duplicated entry point; the default is CAUTION. The loader generates the duplicate entry error message with the severity level you specify. See "Program Duplication and Block Assignment," page 75, for more information on duplicated entry points.

Format:

```
DUPENTRY=keyword1[,keyword2[,keyword3]]
```

FATAL, WARNING, CAUTION, NOTE, COMMENT

See the descriptions for these in "MLEVEL directive," page 39.

IGNORE This is the same as COMMENT.

The default for segldr is DUPENTRY=CAUTION,CAUTION,NOTE. The default for ld is DUPENTRY=CAUTION,NOTE,NOTE.

The first keyword controls the severity level of messages issued for cases in which both duplicated entry points are in a bin file. The second keyword controls the severity level of messages issued for cases in which one duplicated entry point is in a bin file and the other is in a lib file. The third keyword controls the message severity level for cases in which both duplicated entry points occur in a lib file. Table 4 shows this correspondence.

Table 4. DUPENTRY keywords for duplicated entry definitions

Keyword	bin file	lib file
keyword$_1$	both entries	N/A
keyword$_2$	one entry	one entry
keyword$_3$	N/A	both entries

If the second or third keyword is not provided, the value of the last keyword present is used.

DUPLOAD *directive*
3.5.5

The DUPLOAD directive lets you control the severity of messages that the loader generates when a common block is initialized by two or more modules. The loader generates messages with the severity level you specify with DUPLOAD. This level applies to common blocks referenced by C language modules. The level of messages generated for multiple common block initialization by Fortran modules is one severity level lower than the level you specify. Subsequent initializations of a common block overwrite any preceding ones.

Format:

```
DUPLOAD=keyword
```

FATAL, WARNING, CAUTION, NOTE, COMMENT

See the descriptions for these in “MLEVEL directive,” page 39.

IGNORE This is the same as COMMENT.

NODUPMSG *directive*
3.5.6

The NODUPMSG directive suppresses messages about duplicated entry points. If you know that one or more particular entry points are duplicated, and do not want the loader to issue messages about those symbols, use NODUPMSG to suppress the messages.

Format:

```
NODUPMSG=epname[,epname,...]
```

epname Name of an entry point for which no message should be issued.

NOUSXMSG ***directive*** 3.5.7

The NOUSXMSG directive suppresses messages concerning unsatisfied external references. If you know that one or more specific external references will not be satisfied in your program, and you do not want the loader to issue messages about those references, use NOUSXMSG to suppress the messages.

Format:

```
NOUSXMSG=epname[,epname,...]
```

epname Name of an entry point for which no unsatisfied references are present and for which no messages should be issued.

MSGLEVEL ***directive*** 3.5.8

The MSGLEVEL directive lets you set the severity level for any message that the loader issues. For instance, you can increase the severity for certain cases and decrease the severity for others. If you increase the severity to equal to or greater than the WARNING level for a particular error, the loader will not make your program executable if that error occurs. If you decrease the severity to equal to or below the NOTE level for a particular error, the loader will not print a message if that error occurs.

Format:

```
MSGLEVEL=number:keyword[,number:keyword...]
```

number Number of message for which the severity level should be changed.

keyword FATAL, WARNING, CAUTION, NOTE, COMMENT

See the descriptions for these in "MLEVEL directive," page 39.

Example:

```
MSGLEVEL=268:NOTE,114:WARNING
```

In this example, the severity level of message number 268 is set to NOTE; the severity level of message number 114 is set to WARNING.

Message numbers are displayed as part of every error message that is issued. They are appended to the ldr message group identifier. In the following message example, the message number is 112:

```
ldr–112 sldr:  WARNING
     File ’a.out’ is not executable due to
     previous errors.
```

Controlling entry points and execution 3.6

The XFER, EQUIV, and SET directives let you control the point at which your program begins executing, and they also intercept definitions of entry points at load time.

XFER *directive* 3.6.1

The XFER directive specifies the transfer entry point for your program. Control is passed from the system start-up routine to the XFER entry point. If you do not use the XFER directive, the loader uses the first primary entry point it encounters as the transfer entry point. A primary entry point can be specified by the Fortran language PROGRAM statement, by the CAL START pseudo instruction, or by the C language procedure name of main.

The XFER directive can also be used to identify which primary entry point to use as the transfer entry when the loader encounters more than one primary entry point.

Format:

```
XFER=entry
```

entry Entry point name.

EQUIV *directive*
3.6.2

The `EQUIV` directive lets the loader substitute a call or reference to one entry point for a call or reference to another entry point.

Format:

`EQUIV=`*epname*`(`syn_1`[,`syn_2`,...,`syn_n`])`

epname Name of a target entry point.

syn_i Name of the entry point to be linked to *epname*.

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

Example:

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

In the preceding code sequence, the calls to A and B are linked to C by the following specification:

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

The module containing entry point C is loaded, but the module or modules containing A and B might not be loaded. The module or modules containing A and B are loaded if they are needed to satisfy other references to other entry points within those modules.

In this example, `EQUIV` has the same effect as using a text editor to replace all occurrences of `CALL A` and `CALL B` with `CALL C`, except that you do not have to recompile or change the source code.

SET *directive* 3.6.3

The SET directive assigns a value to an external entry point. A SET value for the specified entry point takes precedence over a value encountered in the relocatable modules.

Format:

```
SET=epname:value[:mod]
```

epname Specifies the entry point to be given a value.

value Decimal value associated with *epname*.

mod Alignment modifier. *mod* may be one of the following:

W	Represents a word address (default)
P	Represents a parcel address
V	Represents a constant

UNSAT *directive* 3.6.4

The UNSAT directive specifies the names of one or more unsatisfied external references that are placed in the loader symbol tables before loading any object files. UNSAT is useful if all files to be loaded are lib files. Modules from lib files are included in the executable program only if an entry point in the module satisfies a reference to the lib file. With the UNSAT directive, you can specify the entry points that will cause modules to be included from library files.

Format:

```
UNSAT=epname1[,epname2...]
```

$epname_i$ Name of an unsatisfied entry point.

Example:

```
UNSAT=blocktwo
LIB=mylib.a
```

An unsatisfied external reference to blocktwo is generated, causing the module in mylib.a that contains the blocktwo entry point to be included in the program.

Command-line equivalent: –u option

Program alignment and initialization
3.7

The ALIGN, PRESET, and ORG directives let you initialize uninitialized data areas and control the loading of some modules or common blocks.

ALIGN *directive*
3.7.1

The ALIGN directive controls the starting locations of modules and common blocks. The loader recognizes an align bit for each relocatable module and common block containing an ALIGN pseudo-op. See the *Cray Assembly Language (CAL) for Cray PVP Systems Reference Manual*, publication SR–3108.

Format:

```
ALIGN=keyword
```

IGNORE Allocates the local blocks of each module and each common block at the beginning of the word following the previous local block or common block. The align bit is ignored.

MODULES Allocates the local blocks of each module containing code to an instruction buffer boundary according to the instruction buffer size of the machine. The instruction buffer size is 32 words for Cray PVP systems. Common blocks are forced to instruction buffer boundaries only when the align bit is set.

`NORMAL` Allocates the local blocks of each module and each common block with the align bit set to an instruction buffer boundary, according to the machine’s instruction buffer size. The instruction buffer size is 32 words for Cray PVP systems.

If the align bit is not set for a local or common block, that local or common block is allocated at the word following the previous local or common block (default).

Command-line equivalent: `–a` option

`PRESET` *directive* 3.7.2

The `PRESET` directive specifies a value that the loader uses to preset uninitialized data areas within the object module (for example, variables in labeled Fortran common blocks with no `DATA` statements).

Stack-allocated data is not part of the program image. As a result, the loader cannot preset variables that reside on the stack.

Format:

```
PRESET=keyword
```

`ONES` Sets uninitialized data words to `–1`.

`ZEROS` Sets uninitialized data words to `0` (default).

`INDEF` Sets uninitialized data to `O’0605054000000000000000`. This value generates a floating-point error if used as an operand in a floating-point operation.

`–INDEF` Sets uninitialized data to `O’1605054000000000000000`. This value is the same as that of `INDEF`, except that it is negative.

`INDEFA` Sets uninitialized data to the sum of a logical OR operation of `O'0605054000000000000000` and the address of the word being preset. This value is the same as that of `INDEF`, except the address of the word referenced appears in the low-order bits of the value.

`–INDEFA` Sets uninitialized data to the sum of a logical OR operation of `O'10605054000000000000000` and the address of the word being preset. This value is the same as that of `–INDEF`, except the address of the word referenced appears in the low-order bits of the value.

value Inserts a 16-bit, user-supplied octal value into each parcel of uninitialized data words. The *value* must be in the range `0` <= *value* <= `O'177777`

Command-line equivalent: `–f` option

`ORG` *directive*
3.7.3

The `ORG` directive sets the initial addresses for different portions of your program. Normal programs must have `ORG` values of 0. The `ORG` directive should be used only for special-purpose programs.

Format:

```
ORG=corg:dorg:lorg
```

corg Specifies an octal value between 0 and 77777777. The default is 0, which is the initial address for the code portion of the program.

dorg Specifies an octal value between 0 and 77777777. The default is 0, which is used for initial data for the program.

lorg Specifies an octal value between 0 and 177777. The default value is 0.

Miscellaneous global directives
3.8

The SYMBOLS directive determines whether the debug symbol table should be constructed. The KEEPSYM and HIDESYM directives determines the visibility of externals in the relocatable module. The CASE and CPUCHECK directives control case conversions and machine characteristic checking, respectively. The COMPRESS directive controls compression of executable files. The LOGFILE directive specifies the name of the file to which the loader writes log messages. The LOGUSE directive specifies the names of the object or library files for which log messages should be generated.

SYMBOLS *directive*
3.8.1

The SYMBOLS directives determines whether the loader constructs the debug symbol table for the executable program.

Format:

```
SYMBOLS=ON|OFF
```

ON The loader writes symbol table information to the executable file, following the executable program (default).

OFF Instructs the loader to ignore all symbol table information.

Command-line equivalent: –g and –s options

KEEPSYM and HIDESYM *directives*
3.8.2

The KEEPSYM and HIDESYM directives determine the visibility of externals in the relocatable module. By default, global symbols are visible. The HIDESYM directive hides selected symbols. The KEEPSYM directive hides all symbols except the selected symbols. A directive is needed for each symbol affected.

The KEEPSYM and HIDESYM directives are mutually exclusive.

Format:

```
HIDESYM=symbol:type
  KEEPSYM=symbol:type
```

symbol Name of a symbol.

type The type of symbol, which is one of the following:

C Common block symbol

E External symbol

B Both types of symbols (a C language external)

CASE *directive* 3.8.3

The CASE directive controls whether characters in the directives file are converted to uppercase before they are processed.

Format:

```
CASE=keyword
```

UPPER Directs the loader to convert all module, entry point, and common block names in the directives to uppercase. Usually this is desirable when no relocatable modules with lowercase names are encountered.

MIXED Specifies that no translation is done, and names must match exactly (default).

CPUCHECK *directive* 3.8.4

The CPUCHECK directive controls whether machine characteristic checking is done within the loader. Turning off checking allows a slight increase in the execution speed of the loader, but it also allows the loading and execution of modules that have incompatible characteristics.

Format:

```
CPUCHECK=keyword
```

ON Enables machine characteristic type checking (default).

OFF Disables machine characteristic type checking.

COMPRESS *directive*
3.8.5

The COMPRESS directive enables or disables compression of executable files, and it specifies the size of blocks the loader should consider for compression. As the loader loads your program, it scans for large areas of the program in which each word contains the same value. When it finds a block of words with the same value, it generates a compression entry rather than the actual code. The system start-up routine expands the compression entry into actual code at run time. To be eligible for compression, a block must satisfy the following requirements:

- It must contain only data
- It must contain repetitively initialized values
- It must have a block size larger than the compression threshold

Executable programs that have been compressed require less memory to link, as well as less storage space. Execution time of the system start-up routine increases for compressed programs, but file transfer time is decreased.

Format:

```
COMPRESS=keyword
```

OFF Disables all compression.

number Sets compression block size to *number*. The default is 1000.

`LOGFILE` ***directive*** 3.8.6

The `LOGFILE` directive specifies the name of the file to which the loader writes log messages. (See “`LOGUSE` directive,” page 53, for information on log messages.) Normally, this directive should be used in the default `def_seg` and `def_ld` directives files to identify the log file for all users.

Format:

`LOGFILE=`*file*

file Name of a file to which the loader writes log messages.

The log file must be created prior to loader execution, and it must have `write` permission enabled for all users. On systems with multilevel security (MLS), the log file must be created in the most restrictive partition of the file system, so that all users can write to the file. The loader appends log messages to the end of the file; it does not initialize, summarize, or report on the contents of the log file. If the log file is not present, or the loader cannot write to it, the loader suppresses all log messages without issuing an error message.

Command-line equivalent: none

`LOGUSE` ***directive*** 3.8.7

The `LOGUSE` directive specifies the names of object or library files for which log messages should be generated. Normally, this directive should be used in the default directives files `def_seg` and `def_ld` to log the usage of specific object or library files by all users. If the specified library is not a default library (even if it is in a default search path) you should specify the full path name of the library.

Format:

`LOGUSE=`file_1`[,`file_2`,...]`

file Name of an object or library file whose usage should be logged.

Whenever the file specified on the `LOGUSE` directive is processed by the loader, a log message is appended to the log file (specified by the `LOGFILE` directive).

The generated message is in ASCII characters, and it is terminated by new-line characters (“`\n`”). Individual fields within the message are separated by a vertical bar (“`|`”). The message format is as follows:

`loguse|`*filename*`|`*date*`|`*time*`|`*uid*`|`*code*`\n`

filename Name of file.

date Date of reference to the file; format is *mm*/*dd*/*yy*.

time Time of reference to the file; format is *hh*:*mm*:*ss*.

uid User ID of user referencing the file.

Character indicating the type of reference.

`s` The file was scanned, but no modules were included.

`i` The file was scanned, and modules were included.

Command-line equivalent: none.

Introduction to Program Segmentation [4]

When using the loader, you specify the segment structure and the contents of the segments to be loaded. This section describes the principles of the loader program segmentation. The information in this section does not apply to nonsegmented programs. “Examples,” page 123, contains an example of a segmented program.

In addition to automatic segment loading and unloading, the loader lets you do the following:

- Modify the segmentation structure, usually without recompilation.
- Overlay different modules (subroutines) without making significant source code changes.
- Define the contents of a segment by specifying only one module per segment.
- Pass arguments between subprograms residing in different segments.
- Unload segments and any contained data blocks. The loader then reloads the blocks with their updated images.

SEGLDR segment tree concept
4.1

The loader arranges program segments in a tree structure, as shown in Figure 1, page 56. A nonsegmented program consists of only 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.

The *root* segment of a tree is the 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) means moving away from the root segment, and up the tree or branch means moving toward the root segment.

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

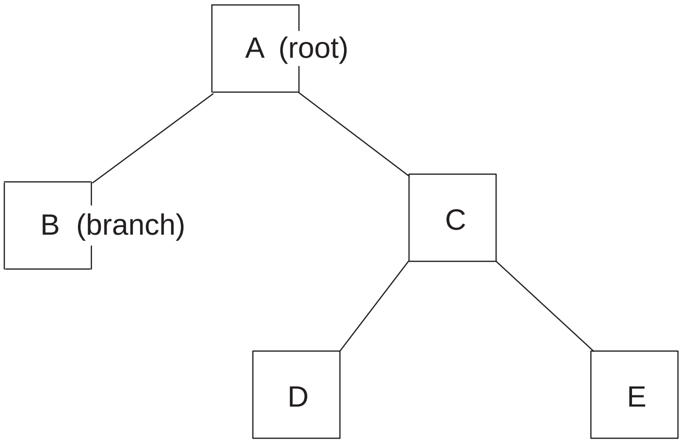


Figure 1. Segment tree

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

The apex of the loader segment tree (segment A in Figure 1) is the root segment. The remaining segments (B, C, D, and E) are the branch segments.

Within these branch segments, B, C, D, and E are successor segments of A. B and C are immediate successor segments of A, and D and E are immediate successor segments of C. It follows, then, that C and A are predecessor segments for D and E, and A alone is the predecessor segment for B and C. C is the immediate predecessor segment of D and E.

Loader segment tree design

4.2

The only restriction on the height or width of the segment tree is that no more than 1000 segments, including the root, can be defined. A valid segment tree, however, must adhere to the following rules:

- 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 can have only one immediate predecessor segment.

Figure 2 and Figure 3 show valid segment trees.

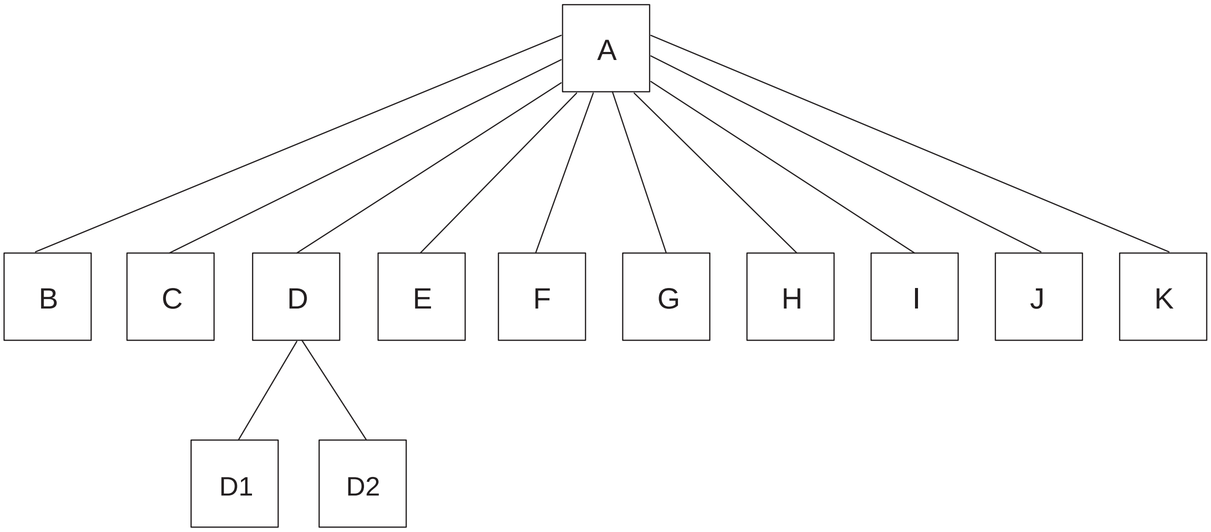


Figure 2. Valid segment tree (broad)

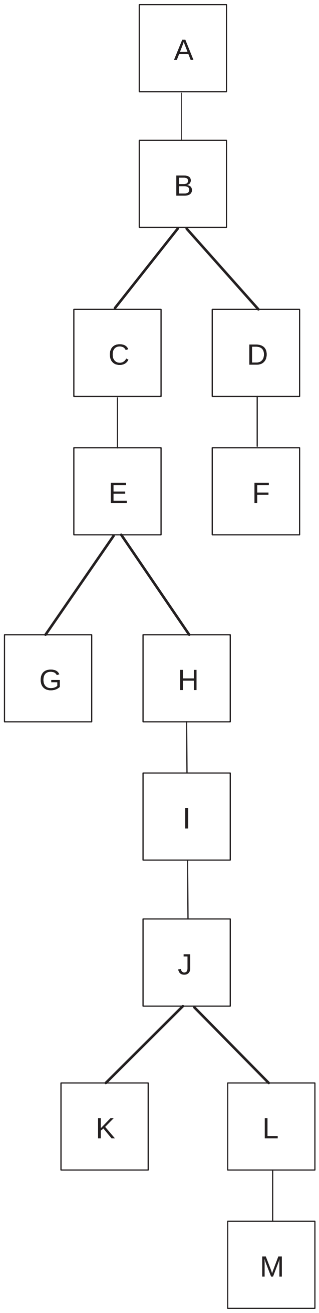


Figure 3. Valid segment tree (deep)

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

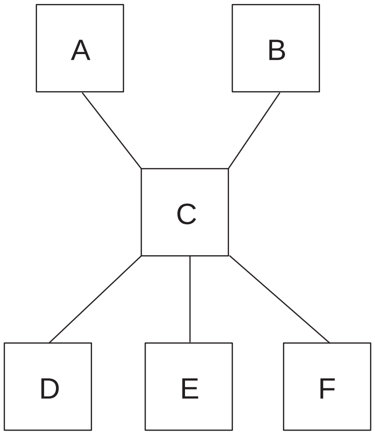


Figure 4. Invalid segment tree (multiple root segments)

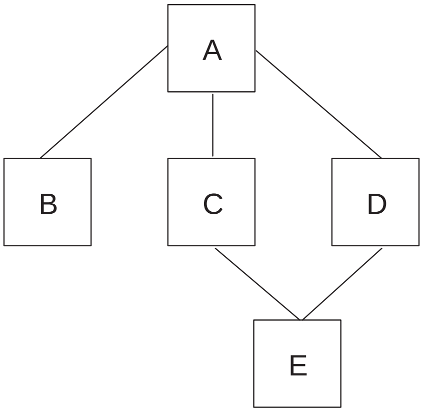


Figure 5. Invalid segment tree (multiple immediate-predecessor segments)

Subroutine calling between segments
4.3

Calls can be made from any module in a segment to any module (subroutine or function) in a successor or predecessor segment. Calls across the segment tree are invalid (see Figure 6, page 61). That is, subroutine calls can be made both up and down the tree if the calling and called modules are owned by segments on a common branch. If a call is made to a subroutine from a segment that is not an immediate predecessor to the segment containing the subroutine, all intermediate segments on the branch are read into memory. In Figure 3, for example, if a line of code in segment I makes a call to a subroutine in segment M, segments J, L, and M are all read into memory.

When a call is made from a subroutine to a subroutine further down the branch at execution time, the loader does the following:

1. Intercepts the call
2. Loads the appropriate segment or segments (if not already in memory)
3. Jumps to the called entry point

The loader intercepts only calls to subroutines in successor segments because they are the only calls that can cause a segment to be loaded (if a segment is in memory, all of its predecessors (callers) are already in memory).

Caution: In CAL, it is strongly recommended that you use the `CALL` and `CALLV` macros for subroutine calls to other modules. If you do not do this, the calls between segments may fail, with unpredictable results.

Do not pass an entry point to a subroutine as an argument if the entry point is not in the same segment or a predecessor segment. In Fortran, for example, the following two statements can produce calls to segments not in memory:

```
EXTERNAL SUB1
CALL SUB (SUB1)
```

The segment `SUB1` may not be in memory when this call is made because the loader cannot detect runtime references.

You should not use the segment structure shown in Figure 6, because it generates an execution error (explanation following).

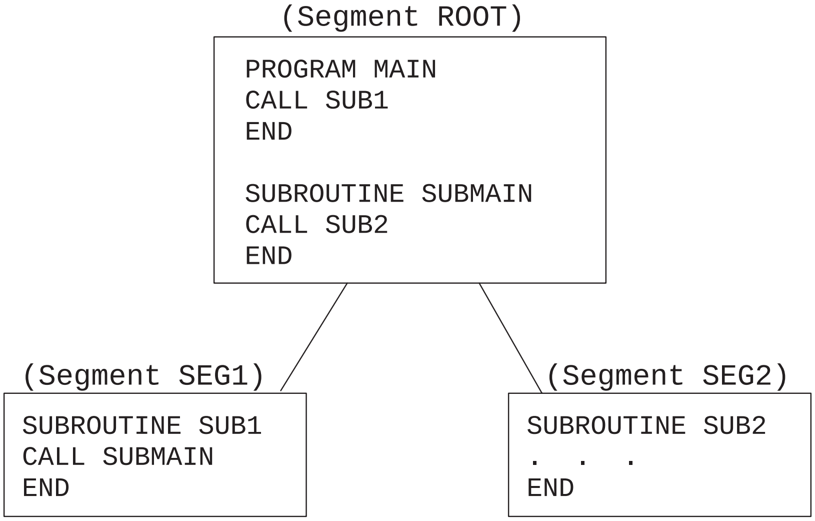


Figure 6. Invalid segment tree (call across segment tree)

Figure 6 shows an invalid segment structure that results in the following sequence of actions:

1. When `SUB1` is called, segment `SEG1` is read into memory.
2. When `SUB2` is called, segment `SEG2` is read into memory, overwriting `SEG1`.
3. On the return to `SUB1` from `SUBMAIN`, `SEG1` is no longer in memory; therefore, control cannot return to `SUB1`.
4. `$SEGRES` terminates the program at this point, displaying an error message.

The loader handles subroutine calls as shown in Figure 7.

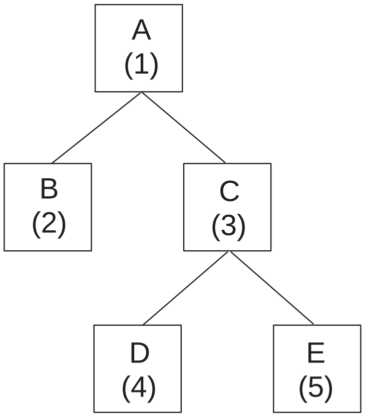


Figure 7. Valid and invalid subroutine references

The following subroutine call descriptions are related to the tree structure shown in Figure 7. Numbers 1 through 5 represent modules in segments A through E.

From	To	Head
1(A)	2,3,4,5	Valid; may need to load some segments.
2	1	Valid; no load needed.
2	3,4,5	Invalid; calls across a branch.
3	2	Invalid; calls across a branch.
3	1,4,5	Valid; may need to load a segment if the call is to module 4 or 5.
4	5,2	Invalid; calls across a branch.
4	1,3	Valid; no load needed.
5	4,2	Invalid; calls across a branch.
5	3,1	Valid; no load needed.

Using segmentation with multitasked programs 4.4

You must be careful when combining segmentation and multitasking in the same executable program, because there is a significant risk of program failure. If a program is multitasked, it is possible for one task to call a subroutine that initiates a segment change, while another task is actively executing in that segment. To avoid this situation, it is necessary to restrict multitasking activity to areas of the program in which segment changes will not occur.

Macrotasking involves partitioning large areas of a program into tasks, so that the tasks can run on several CPUs simultaneously. Because the program tasks contain many subroutines, it is more likely that a segment change will be initiated somewhere within a tasked region of the program. The use of macrotasking in a segmented program is strongly discouraged.

Usually, the tasking activity for an autotasked program is contained within a particular subroutine, although references to other routines are possible. Segment changes are unlikely to occur within tasked regions of the program. If references to other routines are made, you should ensure that all routines within the multitasked region are contained within a single segment.

For more information on multitasking, see the *CF77 Optimization Guide*, publication SG–3773.

Segmentation Directives [5]

This section describes the directives you need for defining the memory tree structure of your program and for assigning modules and common blocks to specific segments. All of the directives in “Segment tree definition directives” and “Segment description directives,” page 66, are segment directives, and they must be placed after all global directives. “Examples,” page 123, contains an example of a segmented program.

Segment tree definition directives
5.1

Use the TREE and ENDTREE segment tree definition directives to tell the loader the shape of the tree that represents the memory layout of your code. Tree structures can be of any width or depth, but they must contain no more than 1000 segments. Only one set of TREE and ENDTREE directives is allowed in a program load.

The TREE directive signals the end of the group of global directives (described in “General Directives,” page 21) and the beginning of the segment tree definition directives. The set of directives specifying the tree structure follows TREE.

The ENDTREE directive terminates the segment tree definition directives; it signals the end of the tree description. The ordering of segment tree definition directives between TREE and ENDTREE is unimportant. The segment description directives immediately follow ENDTREE.

Tree definition directives apply only to segmented programs.

Format:

TREE
segname(*segname*$_1$[,*segname*$_2$,*segname*$_3$,...,*segname*$_n$])
ENDTREE

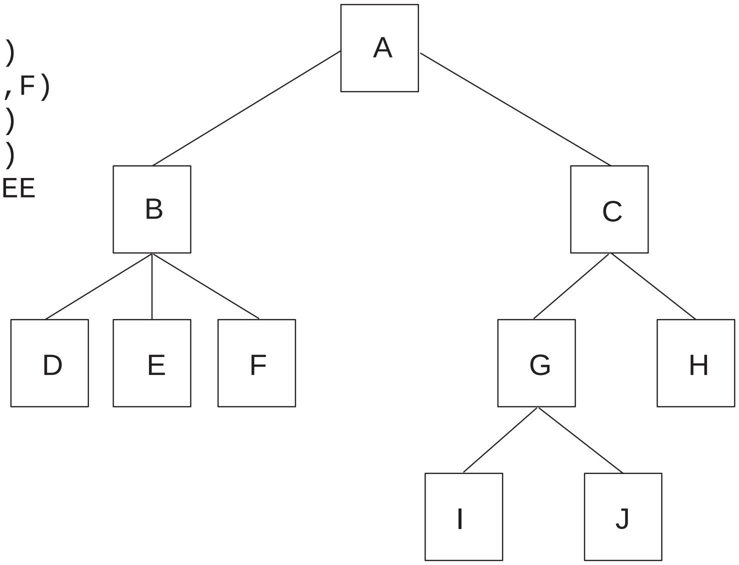
segname Name of a segment.

$segname_i$ Names of all immediate successor segments of *segname*.

If the description of a segment continues beyond one line, end each continued line with a comma.

Example:

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



Segment description directives
5.2

Segment description directives apply only to segmented programs and specify the contents of the segments. At least one module or common block must be assigned to each segment.

In addition to the directives described in this subsection, the `COMMENT`, `ECHO`, and `TITLE` directives discussed in “General Directives,” page 21, can also be used within the segment description directives.

`SEGMENT` *and* `ENDSEG` *directives*
5.2.1

The `SEGMENT` directive specifies the segment being described by the segment description directives. `SEGMENT` is always the first of the segment description directives, except when you are using the `DUP` directive.

The ENDSEG directive terminates the segment description. Any of the segment description directives may appear between SEGMENT and ENDSEG in any order.

Format:

```
SEGMENT=segname
seg descr dirs
ENDSEG
```

segname 1- to 8-character segment name.

seg descr dirs One or more segment description directives.

Example (the // indicates blank common):

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

MODULES *and* SMODULES *directives*
5.2.2

The MODULES and SMODULES directives let you assign modules to the segment specified by the SEGMENT directive. The MODULES and SMODULES directives also order the modules within the segment.

You must assign at least one module to each segment, and you may assign as many as needed. You do not need to assign all modules to segments. “Program Duplication and Block Assignment,” page 75, describes the way the loader handles modules that you have not explicitly assigned to segments. Modules that should be assigned explicitly include those that should reside in the segment specified by the SEGMENT directive but are called by modules in predecessor segments.

If you use the MODULES directive, an error message is issued if the modules specified cannot be located in any included file. Error messages are not issued if SMODULES is used.

Format:

```
MODULES=modname1[,modname2,...,modnamen]
```

modname$_i$ Names of the modules to be loaded.

You may specify argument *modname$_i$* as either *modname* or *modname*:*name*. Use the second form to specify a module to be loaded from a specific file.

If your list of modules is greater than one line, you may use more MODULES directives or end the line with a comma and continue the list on the next line.

Example:

```
MODULES=SUBA,SUBB:lib1.a,SUBC
MODULES=SUBD:FILE.o
```

The loader obtains modules SUBA and SUBC from the first file in which each is encountered. It obtains SUBB from file `lib1.a` and SUBD from file `file.o`.

COMMONS *and* SCOMMONS *directives* 5.2.3

The COMMONS and SCOMMONS directives specify common blocks to be loaded into the segment specified by the SEGMENT directive. Common block specification is optional unless common blocks are to be duplicated or loaded in a specific order.

Common blocks with the same name that are 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, because it is not assigned to a segment. See “Common block use,” page 83, for more information on common blocks.

If you use the COMMONS directive, an error message is issued if the indicated common blocks cannot be located in any included file. No error messages are issued if SCOMMONS is used.

Format:

```
COMMONS=blkname1[:size1][,blkname2[:size2 ],...,blknamen[:sizen]]
```

blkname$_i$ Name of the common blocks to be loaded.

size$_i$ Decimal number indicating the size of the common block. If present, it overrides any common block sizes declared in your code. If the size specified is 0, the first common block size encountered in your code (for this common block) is used. By default, the loader uses the longest common block definition it encounters in your code as the size of the common block.

Common blocks are loaded in the order in which they are specified. The effect of multiple `COMMONS` or `SCOMMONS` directives is cumulative.

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

`BIN` *directive*
5.2.4

The `BIN` directive specifies files containing relocatable modules. The loader loads all modules within the specified `bin` files into the segment specified by the `SEGMENT` directive.

Format:

```
BIN=bin1[,bin2,bin3,...,binn]
```

bin$_i$ Names of files containing relocatable object modules.

The loader processes the files in the order presented. The effect of multiple `BIN` directives is cumulative.

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

Example:

```
SEGMENT=SEG1
BIN=seg1a.o,seg1b.o
BIN=seg1c.o
seg1d.o,seg1e.o
ENDSEG
```

In this example, all modules in files seg1a.o,seg1b.o, seg1c.o, seg1d.o, and seg1e.o are loaded into segment SEG1.

SAVE *directive*
5.2.5

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

Caution: If you do not use the segmented SAVE directive and if you have not specified SAVE=ON as a global directive, SAVE=OFF is assumed. 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 you specify SAVE=ON, however, the loader writes the updated image of the overlaid segment to mass storage before the new segment is loaded. Subsequent execution of a saved segment starts from its saved image. This lets you overlay data areas whose updated values are required in subsequent executions of the saved segment.

Format:

```
SAVE=ON|OFF
```

ON	Enables segment saving.
OFF	Suppresses segment saving (default).

For an example of the use of this directive, see “SAVE directive,” page 72.

DUP *directive*
5.2.6

Use the DUP directive if you want modules with the same name to be loaded into different segments. The DUP directive must precede all SEGMENT directives when duplicate module names are to be loaded.

You can duplicate the modules by using the DUP directive or by using the MODULES directive and assigning the same module name to more than one segment. “Program Duplication and Block Assignment,” page 75, discusses the handling of duplicate modules and entry points in detail.

Format:

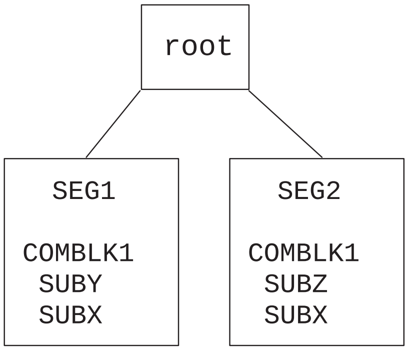
DUP=*modname*(seg_1[,seg_2,...,seg_n])

modname Name of a module to be loaded into more than one segment.

seg_i Names of the segments in which *modname* is to be loaded.

Example:

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



In this example, assume that the module name and entry-point name are the same. Module SUBX is duplicated in segments SEG1 and SEG2. If SUBY is to call SUBX in segment SEG1, SUBY must be assigned to segment SEG1. If SUBZ is to call SUBX in segment SEG2, SUBZ must be assigned to segment SEG2. If SUBY or SUBZ were to go into root, the call would be ambiguous.

Global directives for segmentation
5.3

The directives in this subsection are global directives; that is, they must be specified before the TREE directive and they affect the entire program. These directives apply only to segmented loads.

SLT *directive*
5.3.1

The SLT directive specifies the size of the Segment Linkage table (SLT). The loader's resident run-time routine uses the SLT to service intersegment subroutine calls. The loader writes the actual SLT requirement to the listing file upon load completion. If SLT specifies a size less than the actual requirement, an error message specifies the actual requirement.

Format:

```
SLT=nnn
```

nnn Size (decimal word count) to be reserved for the SLT.

By default, the loader computes the size of the SLT according to the following formula: SLT=40*NBRNCH; NBRNCH is the number of nonterminal segments (segments having at least one successor segment). Calls to predecessor segments need no resident loader intervention.

SAVE *directive*
5.3.2

The global SAVE directive determines whether the current segment states are written to mass storage before they are overlaid with another segment. The global SAVE directive suppresses or enables saving of all segments, but the local SAVE directive can override the global SAVE directive for individual segments.

When SAVE=ON, the loader writes the updated image of the overlaid segment to mass storage before the new segment is loaded. Subsequent execution of a saved segment starts from its saved image; this lets you overlay data areas whose updated values you require in subsequent executions of the saved segment.

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.

Format:

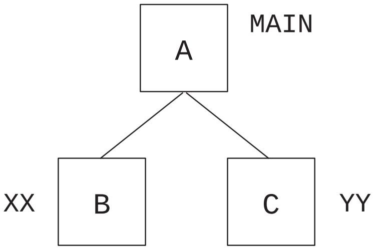
```
SAVE=ON|OFF
```

`ON` Enables segment saving.

`OFF` Suppresses segment saving (default).

Example:

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



The preceding example program performs calculations on two large data arrays, `X(100000)` and `Y(100000)`, contained in subroutines XX and YY, respectively. It completes part of the calculations on one array, then on the other, then returns to the first, and so on, alternating between them. Because the arrays are in two separate subroutines that are never active at the same time, the two arrays can be overlaid rather than forced to the root segment (A).

COPY ***directive***
5.3.3

The COPY directive forces your program to execute from a scratch file. This enables $SEGRES to use a faster form of I/O, which may speed program execution, but increase program start-up time. Programs in which the same segments are loaded and executed many times may improve their performance.

COPY has no effect if SAVE=ON for any segment, because SAVE also forces the use of a scratch file.

Format:

```
COPY=ON|OFF
```

ON Program executes from scratch file, using a faster I/O method.

OFF Disables execution from scratch file (default).

SEGORDER *directive* 5.3.4

The SEGORDER directive lets you determine the order of the segments in an executable file. Ordering the segments can speed up program execution, particularly when part of the file can be contained in buffer memory.

Format:

SEGORDER=seg_1,seg_2,...,seg_n

seg_i Name of a program segment.

The loader writes the segments to the executable file in the order specified. The root segment is always first, regardless of the SEGORDER specification. You do not need to specify all program segments in the SEGORDER directive; segments not specified follow the specified segments in the order in which they are specified in the directives.

Program Duplication and Block Assignment [6]

This section describes two related topics, duplication and block assignment. Duplication occurs when more than one module, entry point, or common block has the same name. The loader handles duplication differently for segmented and nonsegmented programs. Block assignment refers to the process the loader uses to position all modules and common blocks that you have not explicitly assigned.

Duplication and block assignment in nonsegmented programs 6.1

In a nonsegmented program, there is no duplication of modules, entry points, or common blocks.

Duplicate module names 6.1.1

In a nonsegmented load, you can load modules with duplicate names, although this is not recommended because it may result in misleading entry-point definitions, load maps, and debugging.

You can use the MODULES directive with a file specifier to make the loader load a module from a particular file.

Duplicate entry-point names 6.1.2

In a nonsegmented load, each entry point (external definition) must have a unique name. The loader uses the entry point defined in the first module loaded and ignores all subsequent entry points with the same name except to issue a warning message (see “DUPORDER directive,” page 32). You can control the printing of duplicated entry-point messages by using the DUPENTRY directive.

Some of a module’s entry points can be used in the load while others are ignored. The EPXRF parameter in the MAP directive causes the loader to print the Entry Point Cross-reference table, which notes all active and ignored (inactive) entry points.

Duplicate common blocks 6.1.3

Only one common block with a particular name is loaded in a nonsegmented load. The loader assumes that all common blocks with the same name are the same common block (this includes common blocks specified in modules that are never called and, thus, are not loaded).

The loader considers a common block's size to be the largest size encountered in the relocatable modules actually included in the program. You can override this size limit by using the `COMMONS` directive.

Block assignment 6.1.4

All modules and common blocks in a nonsegmented program are assigned to the single segment that makes up the program.

Duplication in segmented programs 6.2

In segmented programs, each segment may contain a module, an entry point, and a common block, each with the same name. Duplication can arise from your use of the `DUP`, `MODULES`, and `COMMONS` directives, or it can arise automatically, as a side effect of using the `FLOAT` directive.

Module duplication 6.2.1

You can manually load copies of the same module or different modules with the same name into different segments. Each segment may have only one module of a particular name.

Duplicate a module by using the `DUP` directive or by using the `MODULES` directive to place the duplicated modules in the desired segments. The loader handles duplicated entry-point names automatically, provided that you have duplicated the modules in your directives.

The loader must know where to put all duplicate module names before encountering the modules. Therefore, you must use the `MODULES` directive to assign all duplicate modules and their callers to the appropriate segment.

Entry-point duplication 6.2.2

Every active entry point in each segment must have a unique name. You must assign all modules containing duplicated entry points and all modules referencing duplicated entry points.

A module referencing a duplicated entry point is linked to the entry point in the same segment. If no entry point with the requested name is in the same segment as the calling module, there can be only one entry point with the duplicated name on the branch.

For example, assume that module X in segment B is in dataset BIN1 and that another module X in segment E is in BIN2. Also assume that the module name and the entry-point name are the same, and that W calls the X in segment B, and Y calls the X in segment E.

Common block duplication 6.2.3

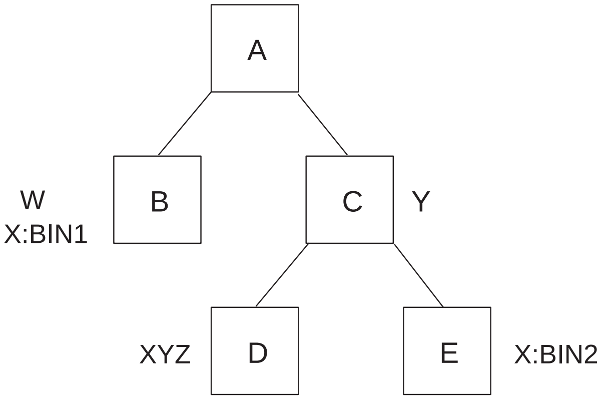
In a segmented load, you can load common blocks with the same name into different segments. Use the COMMONS directive to place the duplicated common blocks in the desired segments. You must also use the MODULES directive to assign every module that references a duplicated common block to the module you desire.

A module referencing a duplicated common block is linked to the common block in the same segment. If there is no common block with the requested name in the same segment as the referencing module, there can be only one common block with the duplicated name on the branch.

Rules for references to duplicated common blocks are the same as the rules for duplicated entry points.

Figure 8 shows the directives required to obtain this description.

```
TREE
  A(B,C)
  C(D,E)
ENDTREE
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
```



Segment assignments:

Segment containing duplicated entry point	Segment calling duplicated entry point	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	C,E	Calls from C are linked to the copy in C. Calls from E are linked to the copy in E.
D,E	A,C	Illegal; both calls are 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.

Figure 8. Entry-point duplication example

Common blocks loaded into different segments are considered unique because they occupy different memory locations. Modules that reference duplicated common blocks must be assigned to different segments to ensure that the program contains no ambiguous references to common block data. (See "`COMMONS` and `SCOMMONS` directives," page 31.)

For example, if common block `/ABC/` were included in segments B and C in the segment tree in Figure 9, a reference to `/ABC/` from a module in segment A would be ambiguous.

In Figure 9, assume that 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.

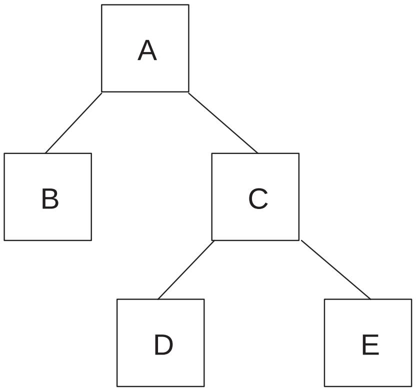


Figure 9. Segment tree with duplicate common blocks

Block assignment in segmented programs
6.3

After you have indicated the segmentation structure and assigned certain modules and common blocks to segments, the loader assigns any remaining movable blocks to segments. A movable block is any module or common block that you have not explicitly assigned to a segment. The loader uses one of two methods to assign movable blocks: floating or automatic duplication. The `FLOAT` directive lets you choose which of these two methods the loader uses.

`FLOAT` *directive* 6.3.1

`FLOAT` is a global directive. It selects the method the loader uses to handle movable blocks.

Format:

```
FLOAT=ANY|NONE
```

`ANY`	Enables movable block floating (default).
`NONE`	Disables movable block floating and enables automatic duplication of movable blocks.

Floating 6.3.2

When floating is enabled, the loader “floats” each movable block up the tree structure to the lowest segment (the one farthest from the root segment) that is a predecessor common to all segments in which the movable block is referenced. The block is thus resident in memory when any segment references it. If the movable block is a common block, all modules that reference it access the same memory space. Floating is the faster of the two methods for loading, and it yields the smallest overall program.

Automatic duplication 6.3.3

When automatic duplication is enabled, the loader assigns a copy of each movable block to each segment that references it, unless a copy of the block has been assigned to a predecessor segment of that block. The block is duplicated automatically in the target segment as if a `MODULES` or `COMMONS` directive had positioned it there. References to a block access unique copies of the block unless it has been assigned to a common predecessor of the modules referencing it. Automatic duplication takes longer to load than floating, and it generates a larger overall program, but it may generate a program that requires less memory to execute. It also allows access to a unique copy of automatically duplicated common blocks.

Example
6.3.4

The following examples show the assignment of movable blocks by floating and automatic duplication. Consider the following partial Fortran program and associated loader directives:

```
PROGRAM EXAMPLE
CALL SUB1
CALL SUB2
CALL ASUB
END

SUBROUTINE SUB1
COMMON /ACOM/ J(200)
CALL BSUB
CALL ASUB
END

SUBROUTINE SUB2
CALL BSUB
CALL SUB2A
CALL SUB2B
END

SUBROUTINE SUB2A
COMMON /BCOM/ I(100)
END

SUBROUTINE SUB2B
COMMON /ACOM/ J(200)
COMMON /BCOM/ I(100)
END
```

Along with this program are the following segmentation directives:

```
TREE
A(B,C)
C(D,E)
ENDTREE
SEGMENT=A
  MODULES=EXAMPLE
ENDSEG
SEGMENT=B
  MODULES=SUB1
ENDSEG
SEGMENT=C
  MODULES=SUB2
ENDSEG
SEGMENT=D
  MODULES=SUB2A
ENDSEG
SEGMENT=E
  MODULES=SUB2B
ENDSEG
```

Figure 10 shows the segmentation structure before movable block assignment.

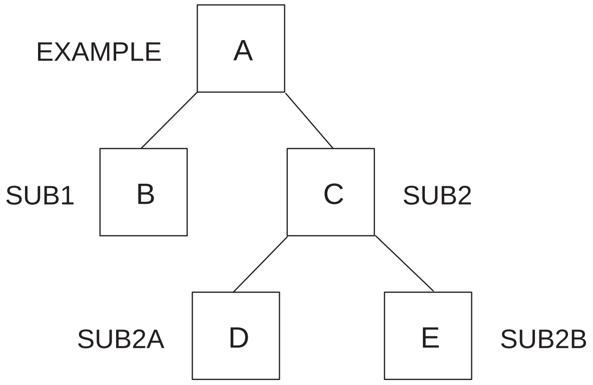


Figure 10. Segmentation structure before movable block assignment

If floating is enabled, the loader makes the following movable block assignments:

- ASUB is assigned to segment A because it is referenced in EXAMPLE.
- BSUB is assigned to segment A to move it to a common predecessor of segments B and C, enabling both SUB1 and SUB2 to reference BSUB.
- ACOM is assigned to segment A to accommodate references to it from SUB1 in segment B and SUB2B in segment E.
- BCOM is assigned to segment C to accommodate references to it from SUB2A in segment D and SUB2B in segment E.

If automatic duplication is enabled, the loader makes the following movable block assignments:

- ASUB is assigned to segment A because it is referenced in EXAMPLE. It is not duplicated in segment B, because ASUB is present in predecessor segment A.
- BSUB is duplicated in segments B and C.
- ACOM is duplicated in segments B and E.
- BCOM is duplicated in segments D and E.

Common block use
6.4

This subsection describes some restrictions that apply to common blocks in segmented programs.

Data load restrictions
6.4.1

Data loads from modules in segments other than the segment in which the common block resides are not processed. The loader issues warning messages for data loads from other segments and skips the data.

The dynamic common, blank common, and task common blocks cannot be data loaded.

Block data routines
6.4.2

The loader always loads modules in `BIN` files that are block data routines. If block data in `LIB` routines is to be loaded, it must be referenced by a previously loaded program (using an `EXTERNAL` statement in Fortran) or by the loader's `MODULES` directive.

If you have a subroutine (not block data) that is never called but contains data loads, you can use the `MODULES` and `FORCE` directives to ensure that it is loaded.

Referencing data in common blocks
6.4.3

Data in a common block can be referenced by any module in either the same or a predecessor segment.

⚠ **Caution:** Referencing a common block that is in a successor segment is not recommended, because it is not guaranteed that the successor segment is memory resident at the time of the reference. This can cause unpredictable and incorrect program results.

Segmented Program Execution [7]

Segmented programs are called into execution in the same manner as are nonsegmented programs. Additional control statement parameters can be provided.

$SEGRES 7.1

On execution, the operating system transfers control to the $SEGRES routine. $SEGRES is a system routine that resides in the loader and is loaded with the object module. It reads segments into memory for execution and writes segments to mass storage to save current segment states.

$SEGRES accepts control from the operating system when execution begins, and it is responsible for some initialization functions. $SEGRES first determines whether the executable binary code can be read from the executable file every time a segment needs to be loaded. If you specified SAVE=ON, all segments are copied to a scratch file in which all reading and saving are done. You can control the scratch file location by using the TMPDIR environment variable. Control transfers to the main entry point in your program after the copy operation to the scratch file.

$SEGCALL intercepts subroutine calls that might require the loading of called segments into memory. $SEGCALL also saves memory-resident segments if SAVE=ON for those segments; this ensures that they are not overwritten.

At execution time, common block /$SEGRES/ conveys information collected by the loader during the load process to $SEGRES. The information that is passed includes segment sizes and addresses, and addresses of intercepted calls between segments.

Subroutine call overhead
7.2

In a segmented load, there are five types of calls to subroutines. Table 5 describes the overhead needed for each type of subroutine call.

Table 5. Subroutine call overhead

Segment containing called routine	Action taken by $SEGRES
Same segment as calling routine	The call is not intercepted.
Predecessor segment of calling routine	The call is not intercepted.
Successor segment in memory	After determining that the segment is resident, control transfers to the called routine.
Successor segment not in memory	One or more successor segments are read into memory; then control is transferred to the called routine.
Successor segment not in memory and SAVE=ON	One or more currently-resident segments are written to a scratch file so that they are not overwritten when the called segment is read into memory. After the needed segments are read into memory, control transfers to the called routine.

Dynamic Memory Management [8]

The loader supports the following two types of dynamic memory management:

- The HEAP, STACK, and TSTACK directives let you use dynamic memory managed by the system heap routines. The ADDBSS directive lets you expand the initial size of your program and reserve space for later memory expansion.
- The DYNAMIC directive lets you specify a common block that can be expanded or contracted at your discretion.

You can use either one or both of these schemes in a single program.

All directives in this section are global directives.

Managing global heap memory
8.1

The HEAP, STACK, and TSTACK directives let you control the size and location of the system-managed heap and stack. Memory space can be acquired from the heap by use of the system heap routines. Under the UNICOS operating system, the heap is always present and resides after the longest segment branch of your program. Heap space is available to all segments of your program.

HEAP *directive*
8.1.1

The HEAP directive allocates memory that the heap manager can manage dynamically. All memory requests are satisfied with space from a common heap. The HEAP directive allows the memory use within a job to increase.

The heap is located in memory following the segment tree branch that occupies the largest amount of memory. HEAP has the same effect on both segmented and nonsegmented programs.

Format:

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

init Initial number of decimal words available to the heap manager; the default is specific to each system.

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, the loader ignores an increment size other than 0. The default is specific to each system.

Command-line equivalent: –H option

STACK *directives*
8.1.2

The STACK directive allocate part of heap memory to a stack for use by re-entrant programs. When you use STACK, the HEAP directive is not needed unless you want to change the default heap values.

The STACK directive is intended for use by individual users to set the stack size for their programs. The following paragraphs outline the steps the loader takes in determining a program's stack size:

1. If a STACK directive has been used, the initial value specified with the STACK directive becomes the program's initial stack size.

2. If no STACK directive is present, the loader analyzes the module calling structure of the program. It estimates what the stack requirements of the program will be. Run-time characteristics of the program, such as regression or indirectly invoked procedures, can cause the estimate to be inaccurate. The loader may underestimate the stack requirements needed for execution. The loader rarely overestimates program stack requirements.

3. If a DEFSTACK directive, see page 118, has been encountered, the loader will compare the estimated size with the initial value specified with the DEFSTACK directive. The larger of the two values will be used as the initial stack size of the program.

4. If no DEFSTACK directive is present, the loader will use the estimated value as the initial stack size.

Format:

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

init Initial size, in decimal words, of a stack. If *init* is less than or equal to 128 words or is absent, an installation-defined value is used.

inc Size, in decimal words, of additional increments to a stack if the stack overflows. A value of zero (0) implies that stack overflow is prohibited. An installation-defined value defines the default increment value.

Command-line equivalent: –S option

The use of more than one of the STACK, HEAP, and FREEHEAP directives can easily result in an inconsistent specification. If this occurs, the maximum size heap is used.

TSTACK *directive*
8.1.3

Multitasked programs often have more extensive stack requirements than unitasked programs. Slave tasks often require a different amount of stack space than the main program task. The TSTACK directive allows you to specify a stack size to be used whenever slave tasks are initiated. In the absence of a TSTACK directive, the loader estimates the amount of stack space required for the slave tasks by using the same algorithm that is used to estimate main program stack size.

Format:

```
TSTACK=init[+inc]
```

init Initial size, in decimal words, of the stack space assigned to each slave task when the slave task begins execution.

inc Size, in decimal words, of additional increments to a stack if the stack overflows. A value of zero (0) implies that stack overflow is prohibited. An installation-defined value defines the default increment value.

`ADDBSS` *directive* 8.1.4

The `ADDBSS` directive tells the loader to expand the initial size of your program. This provides preallocated space for later requests of the program to expand its heap space.

Format:

```
ADDBSS=value
```

value The number of 1024-word blocks of space to add to the uninitialized data area of your program.

Command-line equivalent: `-b` option

`DYNAMIC` *directive* 8.1.5

The `DYNAMIC` directive specifies the common block that can expand or contract under your control. You must call the system routines to expand your program size before referencing the portions of the dynamic common block not initially allocated to your program. The common block occupies memory following the largest segment, and all segments have access to it at any time during program execution. The contents of the dynamic common block may not be declared at load or compile time (data loaded).

Format:

```
DYNAMIC=comblk|//
```

comblk Allocates the specified common block to the first word following the longest segment branch. Only one common block can be specified.

`//` Specifies the blank common block as dynamic.

If no `HEAP` is required, blank common is always dynamic (default); otherwise, there is no default dynamic common block.

If you expand a common block that is not the dynamic common block, you may overwrite a segment in memory, or, when the loader brings in the successor segment, the loader may overwrite the common block. Use the dynamic common block instead.

Example:

CFT program

```
PROGRAM X
COMMON /DYNCOM/ SPACE(1)
.                       In this user-supplied code, the
.                       user requests 9999 additional
.                       words of memory.
DO 100 I=1,10000
   SPACE(I)=0           This code zeroes out 10,000 words,
.                       but only 1 word is actually
.                       preallocated by the loader.
100 CONTINUE
```

SEGLDR directive

```
DYNAMIC=DYNCOM          Identifies /DYNCOM/ as the dynamic
                        common block.
```

Using the heap and dynamic common together

8.2

You can use the heap and dynamic common together in a program if you are careful to adhere to the following guidelines: When both the heap and dynamic common are used, the heap begins immediately after the longest segment branch of your program, and it has a fixed size. No expansion of the heap is allowed. The dynamic common block begins after the heap, and it can expand.

Because the heap cannot expand, the initial size assigned to it must be large enough to accommodate all requests for heap space. This is critically important under the UNICOS operating system because many system library routines request heap space to perform their functions. In general, the initial size of the heap should be at least 5000 words.

The following examples use several system routines for memory management. Additional memory management routines are also available. If you require further information about any of the library routines used in these examples, consult the library manual appropriate to the language and operating system.

Fortran example for acquiring space from the heap

8.2.1

The following is an example of a Fortran program that acquires a 1-Mword block of heap space. You do not need loader directives, but you may use some to set heap values to something other than their defaults.

```
    PROGRAM USEHEAP
    INTEGER SPACE(0:0), ERRCODE, INDEX
    POINTER (SPTR,SPACE)

    CALL HPALLOC (SPTR, 1000000, ERRCODE, 0)
    IF (ERRCODE .EQ. 0) THEN
        DO 1 INDEX = 0, 999999
            SPACE(INDEX) = INDEX

1       CONTINUE
    ENDIF
    END
```

Fortran example for using dynamic common 8.2.2

The following is an example of a Fortran program that runs under the UNICOS operating system and sets up a dynamic common block of 1 million words. The example requires the use of SBREAK, a Fortran interface to the system library routine sbreak, documented in the *UNICOS System Calls Reference Manual*, publication SR–2012. SBREAK expands the field length of the program for the additional space. For this example, you also need the two loader directives: DYNAMIC=DYNCOM, to identify the dynamic common block, and HEAP=10000+0, to set up a heap size large enough and to indicate that it cannot expand. Both of these directives are described in this manual.

```
    PROGRAM USEDYN
    COMMON /DYNCOM/ SPACE(1)
    INTEGER SBREAK, ERRCODE
    . Only one word of space is preallocated to the program.
    . The user must call the system library routine SBREAK to
    . expand the program's field length and to acquire the
    . additional space.
    ERRCODE=SBREAK(1000000)
    IF (ERRCODE .GE. 0) THEN
         DO 100 I=1,1000000
              SPACE(I) = 0.0

100      CONTINUE
    ENDIF
    END
```

Central Memory Allocation by SEGLDR [9]

This section describes the different techniques that the loader uses to allocate user code and data into central memory on various Cray Research systems. Generally, you do not need to know about the techniques that the loader uses, because the default for your system is selected to work for most applications. For some applications, you may need to override the loader defaults, and this can be done using the directives described in "Program alignment and initialization," page 47.

If your application depends on any particular memory allocation scheme, it is recommended that you generalize the program to remove this dependency. Such code is nonstandard, and such dependencies can hinder maintenance of the code over time as systems change.

You can use the `ORDER` directive to specify the memory allocation scheme you desire. This works as long as you do not try to run your code on a different Cray Research system that does not support the specific option. Cray Research has changed and added memory allocation algorithms in the past, and will continue to do so, with the aim of improving the ease-of-use, system throughput, and performance of Cray Research systems. Applications that depend on specific memory allocation schemes will likely not be stable over time.

Definitions of terms
9.1

The following terms are used in this section:

Term	Definition
Block	The unit in which compilers and assemblers generate code and data for the loader to load. The actual memory size of a block is determined by the program.
Code block	A block containing nothing but instructions.
Common block	A block equivalent to the entity defined by a Fortran COMMON statement or C global data item.
Initialized ... block	A local data or common block that has initial values assigned by the program (as with the Fortran DATA statement).
Local data block	A block containing statically allocated local data.
Mixed block	A block containing both instructions and local data.
Uninitialized ... block	A local data or common block with no initial values assigned by the program.

Executable program organization
9.2

Every UNICOS executable program is organized in three sections: the text section, the data section, and the BSS section. Normally, the text section contains instructions, the data section contains initialized static data, and the BSS section contains uninitialized static data. Only the text and data sections are written into the executable file. The BSS section of the program is allocated at execution time. The various allocation methods attempt to maximize placing uninitialized blocks into the BSS section whenever allowed by the hardware and the constraints of the allocation scheme. The placement of blocks into either the text or data section is critical only for shared text programs.

ORDER directive
9.3

ORDER is a global directive. It lets you control the central memory allocation method used by the loader.

Format:

```
       ┌              ┐
       │TEXT,DATA,BSS │
ORDER= │   SHARED     │
       │   SS.TDB     │
       └              ┘
```

The operation of each allocation scheme is described in the following paragraphs.

SHARED Separates the program code and data into two distinct address spaces and collates each one. ORDER=SHARED is used to create shared text programs that execute on Cray PVP systems under the UNICOS operating system, but ORDER=SHARED is not allowed on other systems. The program cannot contain any blocks of mixed code and data if this option is to be effective.

TEXT,DATA,BSS
Allocates code (TEXT) blocks, followed by initialized data (DATA) blocks, followed by uninitialized data (BSS) blocks. This is the default.

SS.TDB Creates a split-segment program and allocates code (TEXT) blocks, followed by initialized data (DATA) blocks, followed by uninitialized data (BSS) blocks. See “Memory allocation for segmented programs,” page 99, for more information.

> **Note:** ORDER=SHARED cannot be used with segmented applications. ORDER=SS.TDB cannot be used with nonsegmented applications.

Command-line equivalents: –n and –O options

TEXT,DATA,BSS allocation scheme for memory allocation
9.4

The TEXT,DATA,BSS allocation scheme is the default on Cray PVP systems. The TEXT,DATA,BSS scheme allocates memory in the following order:

1. Code blocks
2. Initialized local data blocks
3. Initialized common blocks
4. Uninitialized local data blocks
5. Uninitialized common blocks

The TEXT,DATA,BSS scheme assigns as many uninitialized blocks as possible to the BSS section of the program.

Shared-text allocation scheme for memory allocation
9.5

The SHARED allocation scheme can be used to create shared text programs. In order to create a shared-text program, all object modules used in the program must be split into fully-separated code and data blocks. All Cray Research compilers generate separate code and data blocks and all Cray Research libraries contain separated modules. If you include your own assembly language routines, however, you must ensure that the generated code is separated from other modules in your program by including CODE and DATA attributes in any SECTION pseudo-instructions. If all modules are separated, the loader loads all the code sections of the program into one address space, and then loads the DATA and BSS sections into a separate address space.

Advantages of shared-text programs 9.5.1

A shared-text program has two major advantages:

- Multiple processes using the same application can share code, while keeping separate data areas. Thus, process demands on central memory are reduced.
- When the UNICOS operating system allocates memory for a process, it must find sufficient contiguous memory to allow the process to execute. With split code and data, the amount of memory required for the process is the same, except it is divided into two smaller pieces. Therefore, the UNICOS operating system can search for two small sections of memory rather than a single large one.

Disadvantages of shared-text programs 9.5.2

A shared-text program has two major drawbacks:

- The CDBX debugger cannot operate on shared-text programs. You should not use the shared-text scheme while debugging the program.
- Shared-text programs cannot be segmented.

Memory allocation for segmented programs 9.6

The allocation orders specified by the `ORDER` directive allocate each segment as a contiguous area of memory. Each segment is allocated separately; the modules and common blocks assigned to the segment are allocated in the specified order. Each segment begins where its predecessor ends.

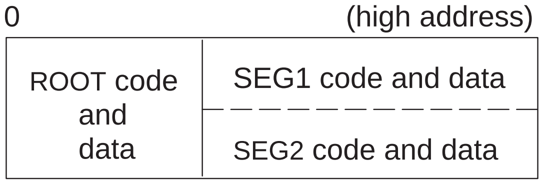
On Cray PVP systems, code must reside in the first 4 Mwords of memory. Large data areas in the root segment may occupy enough memory below these limits to force code in later segments above these limits. To successfully load programs that encounter this problem, you can use the `SS.TDB` value for the `ORDER` directive.

The `SS.TDB` allocation order creates split-segment programs. Each program segment is separated into a data section and a code section, which are allocated separately. Any modules and common blocks assigned to a segment are still allocated in the specified order. The code section of each segment is allocated in memory starting where the code section of the segment's predecessor ends. The data portion of each segment (except the root segment) is allocated in memory following the data section of the segment's predecessor. The data section for the root segment is allocated after the highest address used to store code from the segments.

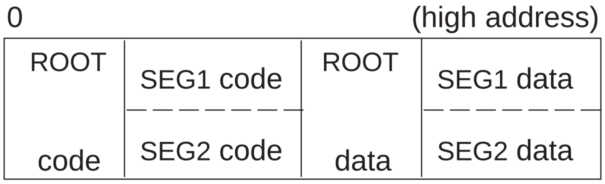
The following segment tree directives describe a program with a root segment and two successor segments:

```
TREE
ROOT(SEG1, SEG2)
ENDTREE
```

The MODULES,COMMONS, COMMONS,MODULES, and TEXT,DATA,BSS allocation orders create a program having the following structure in memory:



The SS.TDB allocation order creates a program with the following structure in memory:



ORDER=SS.TDB

9.6.1

You should use the SS.TDB allocation order on Cray PVP systems when large data areas in the root segment of a program force code in successive segments above the 4-Mword memory boundary. The SS.TDB allocation scheme creates a split-segment program, allocating blocks to the code and data sections within each segment, as follows:

Code section:

- Code and mixed blocks

Data sections:

- Initialized local data blocks
- Initialized common blocks
- Uninitialized local data blocks
- Uninitialized common blocks

Soft Externals [10]

This section describes the special handling that the loader performs when processing “soft” references to an external symbol, or “soft externals.”

Soft external references
10.1

Soft externals let the user control whether modules containing entry points to external functions or data objects are linked to the user’s program. If the user program declares a reference to an external function as “soft,” that reference is not sufficient to ensure that the external function will be included in the program. The function will be included only when referenced elsewhere in the program.

For example, Figure 11 contains two user programs, *flowpgm* and *noflwpgm*. *flowpgm* calls `flowtrace`, performs several functions, and then calls `exit`. *noflwpgm* does *not* call `flowtrace`, but it performs several functions, and then calls `exit`. The `exit` routine is called by both user programs; it processes `exit` calls for programs that call `flowtrace` and for programs that do *not* call `flowtrace`. Therefore, `exit` contains conditional calls to `flowexit`, which is an entry point within the `flowtrace` module. If `flowexit` is declared as a “hard,” or normal external reference in `exit`, all of the `flowtrace` module must be loaded with each user program that calls `exit`, regardless of whether the user program calls `flowtrace`. If `flowexit` is declared as a soft external, then the `flowtrace` module is linked to the user program only when `flowtrace` is referenced. In Figure 11, the `flowtrace` module will be loaded with *flowpgm*, but it will not be loaded with *noflwpgm*.

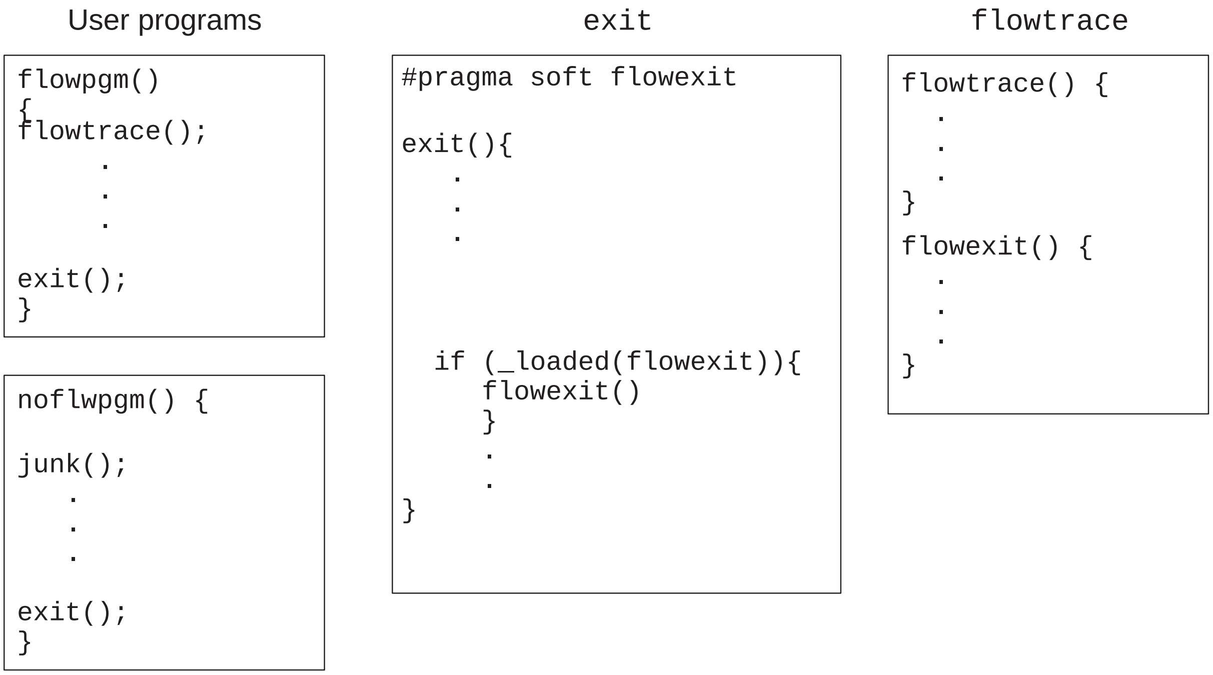
Soft Externals [10]

This section describes the special handling that the loader performs when processing “soft” references to an external symbol, or “soft externals.”

Soft external references
10.1

Soft externals let the user control whether modules containing entry points to external functions or data objects are linked to the user’s program. If the user program declares a reference to an external function as “soft,” that reference is not sufficient to ensure that the external function will be included in the program. The function will be included only when referenced elsewhere in the program.

For example, Figure 11 contains two user programs, *flowpgm* and *noflwpgm*. *flowpgm* calls `flowtrace`, performs several functions, and then calls `exit`. *noflwpgm* does *not* call `flowtrace`, but it performs several functions, and then calls `exit`. The `exit` routine is called by both user programs; it processes `exit` calls for programs that call `flowtrace` and for programs that do *not* call `flowtrace`. Therefore, `exit` contains conditional calls to `flowexit`, which is an entry point within the `flowtrace` module. If `flowexit` is declared as a “hard,” or normal external reference in `exit`, all of the `flowtrace` module must be loaded with each user program that calls `exit`, regardless of whether the user program calls `flowtrace`. If `flowexit` is declared as a soft external, then the `flowtrace` module is linked to the user program only when `flowtrace` is referenced. In Figure 11, the `flowtrace` module will be loaded with *flowpgm*, but it will not be loaded with *noflwpgm*.



```
flowpgm()
{
flowtrace();
     .
     .
     .

exit();
}
```

```
noflwpgm() {

junk();
   .
   .
   .

exit();
}
```

```
#pragma soft flowexit

exit(){
   .
   .
   .



  if (_loaded(flowexit)){
     flowexit()
     }
     .
     .
}
```

```
flowtrace() {
   .
   .
   .
}
flowexit() {
   .
   .
   .
}
```

Figure 11. Soft external usage

How to declare soft externals
10.2

References made to entry points located outside a compilation unit are usually “hard,” or normal references. The assembler (as(1)) and C compiler (Cray C compiler version 5.0 and on and Cray Standard C compiler version 2.0 and on) allow you to declare a reference to be soft.

A soft external in assembly language is declared by using the soft modifier on the ext directive. For example:

```
ext getmsg:soft
```

This statement declares that all references in this module to the external symbol getmsg will be soft references.

To declare a soft external in C, use the #pragma directive, as follows:

```
#pragma soft getmsg
extern int getmsg();
```

The #pragma directive should appear before any references to the external entry point. The directive affects the entire source file.

How to link soft externals
10.3

The loader handles hard and soft references in different ways. If the definition has been found by the loader, hard references to an external entry point are always satisfied by the symbol definition. A hard reference to a library entry point will cause the module containing that entry point to be included in the executable program.

A soft reference is not automatically satisfied by the symbol definition. To satisfy the soft reference, the entry point must be included in the program for some other reason. A soft reference to a library entry point is not sufficient to cause the module containing that entry point to be included in the executable program.

You can cause the library entry point to be included in the program by including one of the following in your program:

- Include hard references to the entry point in the program.
- Include hard references to other entry points in the same module so that the module will be included in the program.
- Force-load the object module. See “Including object modules,” page 24, for a discussion of object module inclusion and force-loading.

As is the case with hard references, if the entry point is included in the program, the soft reference is satisfied by the entry point. If the entry point is not included in the program, the soft reference is converted into an unsatisfied external reference. If the reference has not been satisfied, no error message will be generated indicating that the reference is unsatisfied. If the entry point is referenced during program execution, an appropriate error message will be issued and program execution will terminate.

Using soft externals
10.4

At load time, the loader determines if a soft reference should be linked to the corresponding entry point. An execution-time test is needed to determine whether the reference is satisfied and can be called. You can either use the library routine _loaded, or use a flag word, to perform the test.

Testing entry-point references with _loaded
10.4.1

If the input argument to the library routine _loaded is an entry point that has been included in the program, the library routine _loaded returns a nonzero value.

The following example is a simplified version of the program exit processing, and it illustrates the use of _loaded. The exit routine is called at the end of every program. It needs to call the flowexit routine if flowtrace processing has been enabled; flowexit is contained in the same module as the entry point flowtrace. The flowtrace entry point will be called if the flowtrace processing is enabled; therefore the soft reference to flowexit from exit will be satisfied. If flowtrace is not called, the soft reference to flowexit from exit will not be satisfied. The code in exit.c that calls flowexit takes the following form:

```
#pragma soft flowexit
extern int flowexit();
extern int _loaded();
exit () {
...
  if (_loaded(flowexit))
     flowexit();
}
```

Testing entry-point references with flag words 10.4.2

The second test method uses a flag word rather than the _loaded routine. The following code uses the same example to illustrate how a flag word is used:

```
/* flowtrace.c */

int flowflag = 1;
flowtrace () {
 ...
}
flowexit () {
 ...
}


/* exit.c */

int flowflag;
exit () {
...
if (flowflag)
  flowexit();
}
```

If the module from flowtrace.c is included, flowflag will have a value of 1, and flowexit will be called. If flowtrace.c is not included, flowflag will be 0 and flowexit will not be called.

How to convert soft references to hard references 10.5

The HARDREF loader directive can be used to force the loader to treat all soft references to one or more entry points as hard references. The loader treats all soft references to the specialized entry points as hard references, and it will satisfy the reference if the definition is found. You can use the HARDREF directive to force the satisfaction of a reference even when no other condition would cause it to be satisfied.

HARDREF *directive* 10.5.1

The HARDREF directive specifies one or more entry points that should be included in the load process. Any soft references made to these entry points are converted into hard references.

Format:

HARDREF=*epname*$_1$[,*epname*$_2$...]

epname$_i$ Name of entry point from which all soft references will be converted to hard references.

How to convert hard references to soft references 10.6

The SOFTREF directive can be used to force the loader to treat all hard references to one or more entry points as soft references. The loader treats all hard references to a symbol name as soft references. The module containing the indicated entry point is included in the program only when some other factor causes the inclusion. (See subsection “How to link soft externals,” page 105, for information.)

SOFTREF *directive* 10.6.1

The SOFTREF directive specifies one or more entry points that should not be included in the load process.

The SOFTREF directive should be used with caution, because it can cause references to symbols to remain unsatisfied, for which no loader error message will be issued. If a program does not make a run-time test to determine whether the reference has been satisfied, and the reference is executed at run time, the program terminates in error.

Format:

SOFTREF=*epname*$_1$[,*epname*$_2$...]

epname$_i$ Name of entry point from which all hard references will be converted to soft references.

Configuration Directives [11]

This section describes the directives you use for defining the configuration you want for the SEGLDR environment. These directives are not used in day-to-day activities.

Specifying default directory search lists 11.1

The DEFDIR directive allows you to specify default directory search lists. The LIBDIR directive allows you to add directory names to user directory search lists.

DEFDIR *directive* 11.1.1

The DEFDIR directive specifies default directory search lists. You can specify separate directory search lists for different machine characteristics.

The loader uses the default search list to find files specified on the –l and –j command-line options, and on the LBIN, LLIB, LINCLUDE, and DEFLIB directives. To find the specified files, the loader searches the directories listed in the user directory search list (specified with the –L command-line option or with the LIBDIR directive). If no user search list has been specified, or if the file is not found in any of the user directories, the loader searches the appropriate default directory search list.

Normally, the DEFDIR directive should be used in the default directive files def_seg and def_ld to establish the default search lists for all targeted machines.

Format:

```
DEFDIR[(chars)]=dirname1[, dirname2, ...]
```

chars Specifies a set of machine characteristics, including the primary machine name, logical name, and numeric characteristics. See the `target`(1) command for information on the characteristics that can be specified in *chars*.

dirname Specifies a UNICOS file system directory name.

When a set of machine characteristics is specified on a `DEFDIR` directive, the characteristics are associated with the list of search directories to create a *targeted* search list. If no characteristics are specified, the `DEFDIR` directive creates an *untargeted* search list. You can specify up to 10 `DEFDIR` directives, each with a different set of characteristics. `DEFDIR` directives are not cumulative. If more than one `DEFDIR` directive with the same characteristics has been specified, the directories specified on the latter directive replace those specified on the former. If more than one untargeted search list is specified, the latter directive replaces the former.

The loader determines the target environment of a program from the `TARGET` environment variable (see subsection 2.3.6, page 14, for more information on the `TARGET` variable), or, if `TARGET` is not set, from the main routine of the program. The loader scans the `DEFDIR` targeted search lists in the order specified. If a set of `DEFDIR` machine characteristics does not conflict with the characteristics of the target environment, the associated search list is used as the default search list for the program. If none of the `DEFDIR` characteristics sets matches the target environment, or if no targeted search lists have been specified, the untargeted search list is used.

Initially, `DEFDIR` specifies the `/lib` and `/usr/lib` directories in the untargeted search list and does not specify any directories in the targeted search list.

Example:

```
    defdir(cray–ymp)=/lib/xlib,/usr/lib/xlib
    defdir=/lib,/usr/lib,/usr/local/lib
```

When the target environment of a program is `cray–ymp`, the `/lib/xlib` and `/usr/lib/xlib` directories are searched. When any other target environment is used, the `/lib`, `/usr/lib`, and `/usr/local/lib` directories are searched.

Command-line equivalent: none

`LIBDIR` *directive*
11.1.2

The `LIBDIR` directive adds directory names to the loader's user directory search list, which is used to find files specified on the `–l` and `–j` command-line options, as well as files specified on the `LBIN`, `LLIB`, `LINCLUDE`, and `DEFLIB` directives. The loader first searches each directory in the user search list. If directories have not been specified, or if the file cannot be located in any of the specified search directories, the loader searches the default directory search list for the file. (See "`DEFDIR` directive," page 109, for information on the default directory search list.)

Format:

`LIBDIR=`*dirname*$_1$`[,`*dirname*$_2$`,...]`

dirname UNICOS file system directory name.

You may specify up to 20 directory names. If this directive continues beyond one line, end each continued line with a comma. Multiple `LIBDIR` directives are cumulative. Each directive adds directory names until the limit of 20 is reached.

Example:

```
    LIBDIR=/mydir/lib,locallib
```

The loader adds `/mydir/lib` and `locallib` (relative to the current directory) to the list of user search directories.

Command-line equivalent: `–L` option

The executable program
11.2

The `OUTFORM` directive give you a measure of control over the executable program that the loader produces. You can tell the loader the type of output file to produce.

`OUTFORM` *directive*
11.2.1

The `OUTFORM` directive specifies the type of the output file of the loader. This directive essentially allows you to build a prelinked collection of files with a `.o` extension. Within this collection of files all internal references have been resolved. This feature helps reduce application link time.

Format:

```
OUTFORM=[ABS|REL]
```

`ABS` The output file will have all internal references resolved (default).

`REL` The output file will have internal references resolved at link time.

`ld` command-line equivalent: `–r` (the executable program will have the relative attribute).

It is assumed that the relocatable output will be invoked only with the `ld` command. If you invoke the relocatable output with the `segldr` command, be certain to include the `SYSTEM=STDALONE` directive.

Controlling entry points and execution
11.3

The START and CALLXFER directives let you control the point at which your program begins executing, and they also intercept definitions of entry points at load time.

START *directive*
11.3.1

The START directive specifies the entry point that receives control from the operating system when the program begins execution. For normal programs executing under the UNICOS operating system, the entry point is the system start-up routine. The default directives file specifies the correct entry point for your system. You should use the START directive only when building a special-purpose program.

Format:

```
START=epname
```

epname Name of entry point at which program execution begins.

CALLXFER *directive*
11.3.2

The CALLXFER directive specifies the entry-point name used by the system start-up routine to call your main program. The loader links references to the CALLXFER entry point to the transfer entry point defined by the XFER directive. The default directives file specifies the correct name for your system. You should use the CALLXFER directive only when building a special-purpose program.

Format:

```
CALLXFER=epname
```

epname Symbol name used by the system start-up routine to call the XFER entry point.

Miscellaneous global directives
11.4

The SYSTEM directive specifies under which operating system your program will execute. The INCFILE directive specifies the name of a previously built executable program. The ZSYMS directive controls whether the loader will include the special zzzzzz?? symbols in the load module.

SYSTEM *directive*
11.4.1

The SYSTEM directive selects the target operating system on which your program will execute. The default directives file specifies a SYSTEM value of UNICOS.

Format:

SYSTEM=*keyword*

UNICOS — Sets the target operating system to UNICOS. When SYSTEM=UNICOS is specified, the loader requires that the START and CALLXFER directives are specified, and enables heap and stack processing, enable task common block processing, and adds the _infoblk information block to your program (default).

STDALONE — Sets the target operating system to be undefined. The loader does not require any directive settings and does not perform any special processing. The STDALONE directive should be used only for special-purpose programs.

INCFILE *directive*
11.4.2

The INCFILE directive specifies the name of a previously-built executable program. The loader extracts the symbol information from the file specified with the INCFILE directive. The extracted symbol information is used to satisfy external references and to allocate common blocks for object modules loaded during this invocation of the loader. When used in conjunction with the ORG, SYSTEM=STDALONE, and other directives, a program fragment is built that can execute in the address space of the original program. The original program must do the following actions: call the loader to create the program fragment, provide the memory space, to read the program fragment into its address space, and pass control to it. The executable output produced when `INCFILE` is used cannot be executed independently. The `INCFILE` directive should be used only for special-purpose programs.

Format:

```
INCFILE=file
```

file Name of a file containing a previously linked executable program.

`ZSYMS` *directive*

11.4.3

This directive controls whether the loader will include the special `zzzzzz??` symbols in the load module. The default is `OFF`.

```
ZSYMS=[ON|OFF]
```

`ON` Include the `zzzzzz??` symbols in the load module.

`OFF` Do not include the `zzzzzz??` symbols in the load module.

Command-line equivalent: none.

Zero address directives

11.5

Zero address directives specify a block that is to occupy address zero. When these directives are used the value zero is no longer a valid pointer value. The `ZEROCOM` directive specifies the name of the common block that is to be placed at the zero address of the data space if common blocks precede local blocks; otherwise all three directives and their corresponding assembly modules are to be provided. The `ZERODATA` directive specifies the name of the module that is to be placed at the zero address of the data space. The `ZEROTEXT` directive specifies the name of the module that is to be placed at the zero address of the text space.

ZEROCOM *directive* 11.5.1

The ZEROCOM directive specifies the name of the common block that is to be placed at the zero address of the data space (if the load order is COMMONS, MODULES; otherwise this directive has no effect). The named module must contain only one common data block. If the directive is not present, or if the named module is not found, no special processing for address 0 is done.

The last ZEROCOM directive encountered is the one used; the earlier ZEROCOM directives are ignored.

This directive should only be used in the default directives file.

Format:

```
ZEROCOM=blkname
```

blkname Name of the common block to be loaded.

ZERODATA *directive* 11.5.2

The ZERODATA directive specifies the name of the module that is to be placed at the zero address of the data space. The named module must contain only one local data block. If the directive is not present, or if the named module is not found, no special processing for address 0 is done.

The last ZERODATA directive encountered is the one processed; the earlier ZERODATA directives are ignored.

This directive should only be used in the default directives file.

Format:

```
ZERODATA=modname
```

modname Name of the module to be loaded.

ZEROTEXT *directive*
11.5.3

The ZEROTEXT directive specifies the name of the module that is to be placed at the zero address of the text space. The named module must contain only one local code block. If the directive is not present, or if the named module is not found, no special processing for address 0 is done.

The last ZEROTEXT directive encountered is the one processed; the earlier ZEROTEXT directives are ignored.

This directive should only be used in the default directives file.

Format:

```
ZEROTEXT=modname
```

modname Name of the module to be loaded.

Managing global heap memory
11.6

The DEFHEAP, DEFSTACK, and FREEHEAP directives let you control the size and location of the system-managed heap and stack. Memory space can be acquired from the heap by using the system heap routines. Under the UNICOS operating system, the heap is always present and resides after the longest segment branch of your program. Heap space is available to all segments of your program.

These directives should only be used in the default directives file.

DEFHEAP *directive*
11.6.1

The DEFHEAP directive allocates memory that the heap manager can manage dynamically. When you use DEFHEAP, the HEAP directive is not needed unless you want to change the default heap values.

The DEFHEAP directive is intended for use in the default directives file to establish a minimum heap size for all programs. See the “HEAP directive,” page 87.

Format:

```
DEFHEAP=[init][+inc]
```

init Initial number of decimal words available to the heap manager. If *init* is less than or equal to 128 words or is absent, a value defined when the system is installed is used.

inc Increment size, in decimal words, of a request to the operating system for additional memory if the heap overflows. A value of zero implies that heap overflow is prohibited. A value defined when the system is installed determines the default increment value.

DEFSTACK *directive*
11.6.2

The DEFSTACK directive allocates part of heap memory to a stack for use by re-entrant programs. When you use DEFSTACK, the HEAP directive is not needed unless you want to change the default heap values.

The DEFSTACK directive is intended for use in the default directives file to establish a minimum stack size for all programs. See the “STACK directive,” page 88, for an outline of the steps the loader takes in determining a program’s stack size.

Format:

```
DEFSTACK=[init][+inc]
```

init Initial size, in decimal words, of a stack. If *init* is less than or equal to 128 words or is absent, a value defined when the system is installed is used.

inc Size, in decimal words, of additional increments to a stack if the stack overflows. A value of zero (0) implies that stack overflow is prohibited. A value defined when the system is installed determines the default increment value.

FREEHEAP *directive* 11.6.3

The FREEHEAP directive specifies the minimum amount of free memory available in the heap after the initial stack allocation. The initial heap size will be the sum of the initial stack size and the value specified by this directive.

Format:

```
FREEHEAP=value
```

value The number of words of space to be left free in the heap after allocation of the stack.

The use of more than one of the STACK, HEAP, and FREEHEAP directives can easily result in an inconsistent specification. If this occurs, the maximum size heap is used.

Scanning Directives [12]

When the target machine is a CRAY EL98 or CRAY J90 system, the loader invokes a special scanner to detect and correct potential problems in the program. The problems result from specific instruction sequences that generate unexpected results when the program uses multitasking on a CRAY EL98 system or enables cache memory on a CRAY J90 system. The loader provides two directives that work in conjuction with the scanner.

SCANNER directive
12.1

The SCANNER directive lets you turn the scanner off or on. The default condition is on when the target system is a CRAY EL or CRAY J90 system. If you are targetting your program for one of these systems and do not want your program scanned, add the SCANNER=OFF directive to your load step. If the target is a CRAY J90 system, your program will execute with cache memory disabled. If the target is a CRAY EL98 system and performs multitasking, you may encounter unexpected results.

Format:

```
SCANNER = [ON | OFF]
```

SCANPAD directive
12.2

When processing a segmented program, the scanner will occasionally be unable to locate enough unused memory areas to apply the necessary corrections. Use the SCANPAD directive to add additional unused memory to your program.

Format:

```
SCANPAD = nnnnnn
```

nnnnnn Additional number of words, in decimal, to add to the program.

Examples [A]

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

Basic case

A.1

The Fortran program in this example is compiled, loaded, and executed beginning at entry point START. The loader produces a full load map. Its source is in file `source.f`. The loaded program is nonsegmented.

```
cft77 source.f
segldr –o ftest –M,f –e START source.o > mapfile
ftest
```

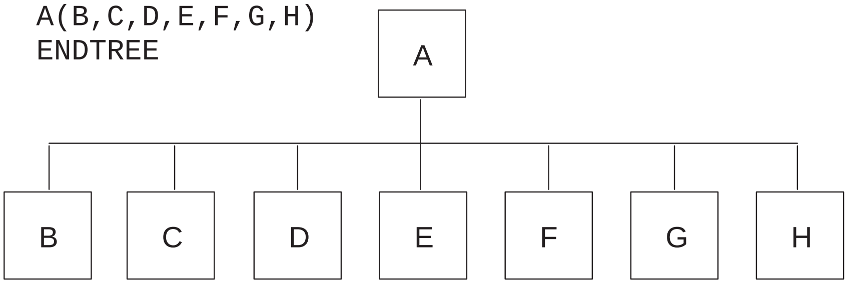
Tree structure examples

A.2

The following two examples show two legal tree structures generated by the loader.

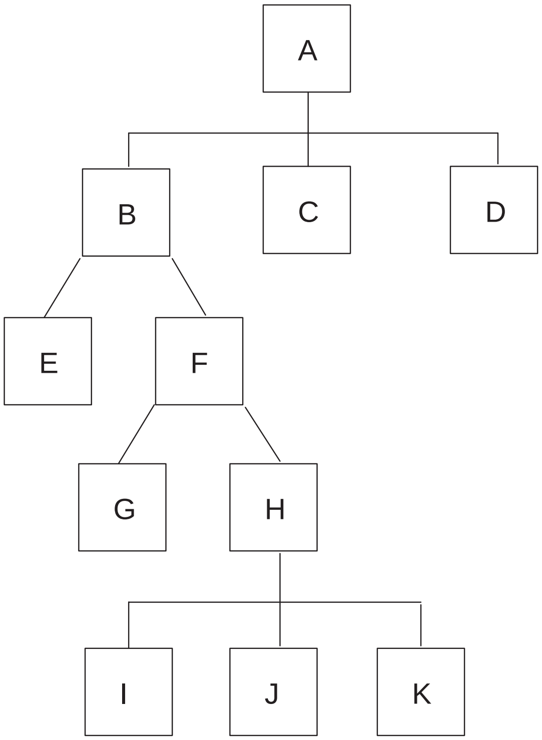
Example 1:

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



Example 2:

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



Tree structure with expandable common block

A.3

Given the tree structure shown in Figure 12, assume that dynamic common block /DYN/ is used and expanded at execution time. All modules are obtained from mybin.o, blib.a, and baselib.a. Common block /AA/ is to be assigned to segment J. A full load map on file map is desired.

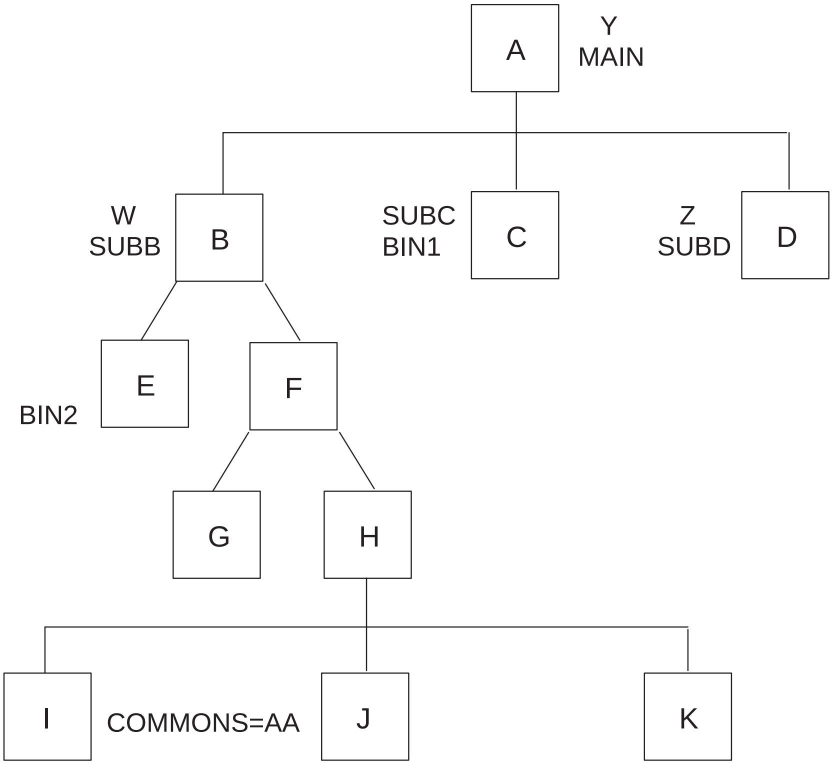


Figure 12. Example tree structure

The control statement and directives required are as follows:

```
segldr –i ins –M map,full –l./blib.a –l./baselib.a mybin.o
```

The following directives are used:

```
DYNAMIC=DYN
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
```

Segmented load with duplicated modules

A.4

This example is based on the tree structure in Figure 13. Given this tree structure, assume that all modules in object file `bin1.o` are to be loaded in segment C and all modules in `bin2.o` in segment E. All other modules are to be obtained from global bin files `bin3.o` and `bin4.o`, and the default libraries. Modules 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 module X (in `bin3.o`) is needed in segment D, and version 2 (in `bin4.o`) is needed in segment F. All calls to entry points Y1, Y2, and Y3 are to be linked to entry point Y. Also assume that the module name and the entry name in a subroutine are the same.

The control statements and directives included are as follows:

```
segldr –i inpts
```

`INPTS` contains the following directives:

```
BIN=bin3.o, bin4.o; 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.o
ENDSEG
SEGMENT=D
MODULES=Z,X: bin3.o
ENDSEG
SEGMENT=E
BIN=bin2.o
ENDSEG
SEGMENT=F
MODULES=X:bin4.o
ENDSEG
```

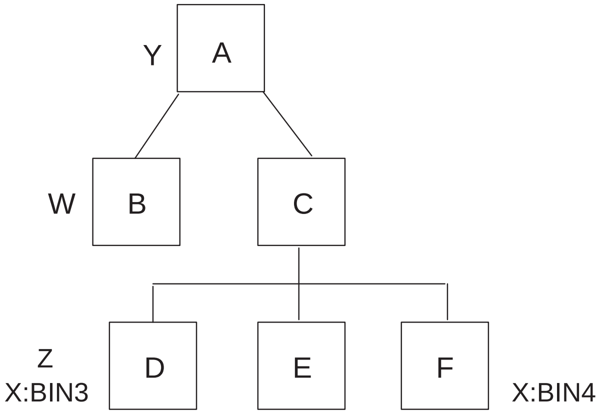


Figure 13. Tree structure

Comprehensive Fortran program example
A.5

This example provides a set of loader directives, block maps and associated output, and related entry point and common block reference maps for the sample Fortran program that follows.

Fortran source code
A.5.1

The following Fortran program consists of 10 subroutines. The loader directives described in the following subsection load the 10 separate modules of this program into separate segments.

```
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
```

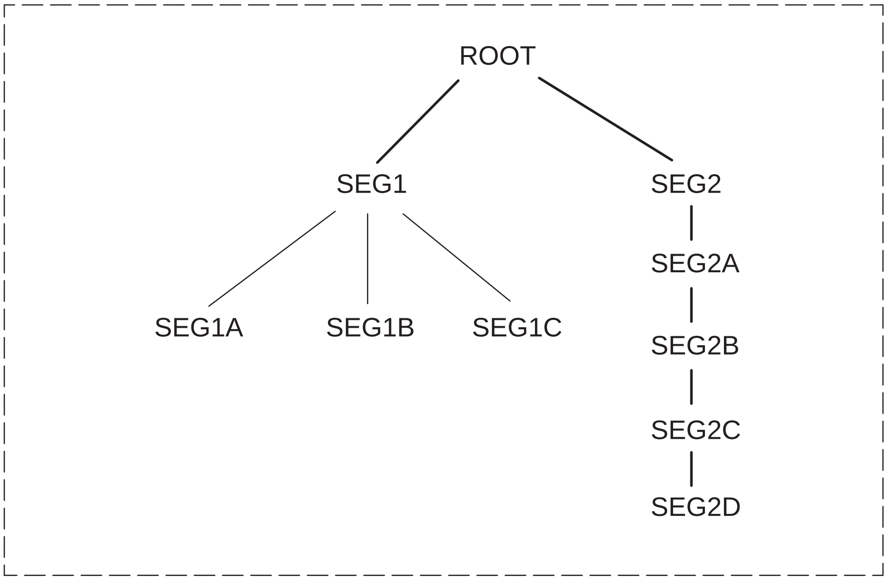
```
SUBROUTINE SUBR1A(I)
COMMON COMMON
PRINT *,’ EXECUTION OF SUBR1A’ I=I+1
RETURN
END

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
```

Loader directives
A.5.2

The following loader directive input sample specifies and diagrams the construction of the segmented object module.

```
ECHO=ON
MAP=FULL
HEAP=5000+0
```



```
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
```

SEGLDR map output
A.5.3

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

```
Program statistics
Segmented object module written to– a.out
Allocation order– XMP.EMA
Movable block positioning– ANY
Actual SLT requirement– 16
Program origin–          0 octal       0 decimal
Program length–     110403 octal   37123 decimal
Dynamic common block– //
        Origin–     110402 octal   37122 decimal
        Length–          1 octal       1 decimal
Maximum segment chain address–       110402 octal   37122 decimal
ending with segment– SEG2D
Transfer is to entry point– EXAMPLE at address– 340a

Managed Memory Statistics
   Initial stack size–                4000 octal    2048 decimal
   Stack increment size–               400 octal     256 decimal
   Initial managed memory size–      11610 octal    5000 decimal
   Managed memory increment size–        0 octal       0 decimal
   Managed memory epsilon–               0 octal       0 decimal
   Base address of managed memory/stack–    76572
   Base address of pad area–                76367
Segment numbers
        0– ROOT     1– SEG1     2– SEG1A     3– SEG1B
        4– SEG1C    5– SEG2     6– SEG2A     7– SEG2B
        8– SEG2C    9– SEG2D
```

Program block maps A.5.4

The segment summary is followed by two block maps for each segment in this example; one sorted by address, and another sorted by block name. This is an abbreviated sample. Library routines have been omitted, and block maps for only the first three segments are present.

```
Segment Summary
Segment Address   Length    Save Histogram (bar =– 884 words decimal)

ROOT          0    75763      -------------------------------------
SEG1       7576       46                                          -
SEG1A     76031       67                                          -
SEG1B     76031       67                                          -
SEG1C     76031        6                                          -
SEG2       7576       34                                          -
SEG2A     76017       73                                          -
SEG2B     76112       73                                          --
SEG2C     76205       73                                           -
SEG2D     76300       67                                           -

Segment ’ROOT’ Block Map – sorted by address
Module  Block    Address   Length   Source       Date
$START                 0       22   /lib/libc.o  02/16/88 07:43
        TRBK          22        7
$SEGRES               31       73   /lib/libu.o  02/16/88 07:46
                     124       57
        CALLIST      203      115
        CALLIST      320        1
        TRBK         321       16
$EXAMPLE             337       27   t/example.o  01/22/87 16:16
        #TB          366        7
        #CL          375       22
        $TRBK        417        7
        /SPACE/    74365      144
        /WAVARS/   74531     1232
```

(continued)

```
Segment ’ROOT’ Block Map – sorted by block name
Module  Block    Address   Length   Source       Date

$SEGRES CALLIST      320        1   /lib/libu.o  02/16/88 07:46
                     124       57
        /$SEGRES/  20633      462
                   21357      363
        CALLIST      203      115
        TRBK         321       16
                      31       73
$START             21317       40   /lib/libc.o  02/16/88 07:43
                       0       22
        TRBK          22        7
$EXAMPLE             337       27   t/example.o  01/22/87 16:16
        $TRBK        417        7
        #CL          375       22
        #TB          366        7
        #DA        61303        7

Segment ’SEG1’ Block Map – sorted by address
Module  Block    Address   Length   Source       Date

SUBR1              75763       17   t/example.o  01/22/87 16:16
        #TB        76002        6
        #CL        76010        6
        $TRBK      76016        7
        #DA        76025        3
        /STATUS/   76030        1

 Segment ’SEG1’ Block Map – sorted by block name
 Module Block    Address   Length   Source       Date

        /STATUS/   76030        1
SUBR1   #CL        76010        6   t/example.o  01/22/87 16:16
        $TRBK      76016        7
        #DA        76025        3
        #TB        76002        6
                   75763       17
```

(continued)

```
   Segment ’SEG1A’ Block Map – sorted by address
   Module Block    Address   Length   Source       Date
   SUBR1A            76031       25   t/example.o  01/22/87 16:16
          #TB        76056       10
          #CL        76066       14
          $TRBK      76102        7
          #DA        76111        7

   Segment ’SEG1A’ Block Map – sorted by block name
  Module  Block    Address   Length   Source       Date

   SUBR1A #CL        76066       14   t/example.o  01/22/87 16:16
          #TB        76056       10
          $TRBK      76102        7
          #DA        76111        7
                     76031       25
```

Program entry-point cross-reference map

A.5.5

This sample entry-point cross-reference map shows entry-point values, segments to which modules are assigned, and the segment tree in caller/callee form.

When you specify MAP=FULL or MAP=EPXRF, this is the resulting output. (This sample is abbreviated for readability.)

```
Entry point references
EXAMPLE from t/example.o in ROOT     calls...  SUBR1 SUBR2 $WLI $WLA
                                               $WLV% $WLF $END
                                               $SEGCALL
    EXAMPLE          340a

 SUBR1 from t/example.o in SEG1      calls...  SUBR1A SUBR1B SUBR1C
    SUBR1          75764a   called by...EXAMPLE

SUBR1A from t/example.o in SEG1A     calls...  $WLI $WLA $WLF
    SUBR1A         76032a   called by...       SUBR1

SUBR1B from t/example.o in SEG1B     calls...  $WLI $WLA $WLF
    SUBR1B         76032a   called by...SUBR1

SUBR1C from t/example.o in SEG1C     calls...  $WLI $WLA $WLF
    SUBR1C         76032a   called by...SUBR1

SUBR2 from t/example.o in SEG2       calls...  SUBR2A
    SUBR2          75764a   called by...EXAMPLE

SUBR2A from t/example.o in SEG2A     calls...  $WLI $WLA $WLF SUBR2B
    SUBR2A         76020a   called by...SUBR2

SUBR2B from t/example.o in SEG2B     calls...  $WLI $WLA $WLF SUBR2C
    SUBR2B         76113a   called by...SUBR2A

SUBR2C from t/example.o in SEG2C     calls...  $WLI $WLA $WLF SUBR2C
    SUBR2C         76206a   called by...SUBR2B

SUBR2D from t/example.o in SEG2D     calls...  $WLI $WLA $WLF
    SUBR2D         76301a   called by...SUBR2C
```

Program common block reference map A.5.6

When you specify MAP=FULL or MAP=CBXRF, the output contains the common block cross-reference.

```
Common Block References
Block   Segment  Address   Length   Module references
$SEGRES ROOT       20633      462   $SEGRES

//               110402         1   SUBR1 SUBR1A SUBR1B

SPACE   ROOT       74365      144   SUBR1 SUBR2
STATUS  SEG1       76030        1   SUBR1B SUBR1C
```

Messages [B]

SEGLDR produces many messages describing problems it detects during the load process. The loader divides messages into two categories:

- Load-time messages produced during the load process. These messages are written to the standard error (stderr) file. By specifying the –k option on the segldr command line, the messages can be forced to the standard output (stdout) file.
- Run-time messages issued when the program executes. the run-time messages are issued by library routines that the loader builds into the load. These messages are always written to the standard error (stderr) file.

The loader produces six classes of load-time messages, five of which can be controlled by users through the use of the MLEVEL directive. From least severe to most severe, the five user controllable message classes are as follows:

Class	Description
COMMENT	Informational messages that have no affect on the execution of the object module.
NOTE	Messages indicate the possible misuse or inefficient use of the loader. These errors have no affect on the execution of the object module.
CAUTION	Messages indicate the possible detection of an error not severe enough to prohibit execution of the object module.

Class	Description
`WARNING`	Messages indicate an error severe enough to invalidate the object module. The object module may not be written, but processing continues so that additional error checking occurs. In most cases, the executable program will still be generated, but execution mode will not be enabled.
`FATAL`	Messages indicate a fatal error was detected and processing cannot continue. No object module is written, the loader terminates immediately.

In addition to these message classes, `segldr` produces `SUMMARY` messages when the `–k` option is specified on the command-line. Unlike the other message classes, `SUMMARY` messages are always written to the standard error (`stderr`) file; they cannot be redirected to the standard output (`stdout`) file or a file through the use of the `–k` option.

`SUMMARY` messages serve as immediate notification that you have errors in your load process.

The loader prepends the type of message onto all messages.

You can get detailed descriptions for any loader error messages through the use of the `explain`(1) command. The loader message ID string for use with the `explain` command is `ldr`.

The following is an example using the `explain` command to generate a message description for the loader message number ldr-101, “`The initial managed memory size is too small.  It has been increased to '`*`nnn`*`' words`:”

```
mjc% explain mppldr101

The initial managed memory size is too small.
It has been increased to 'nnn'
words.

The size specified on the HEAP directive as
the initial managed memory size
is below the minimum value allowed.  The
amount of managed memory has been
set to the minimum size allowed.
```

You can control the format of messages by using the MSG_FORMAT environment variable. For a complete description of the MSG_FORMAT environment variable, see the explain(1) command.

Loader-created Tables [C]

The loader can create and initialize the contents of several tables in the generated program. Four of these loader-created tables, the _infoblk, $SEGRES, Segment Linkage, and Segment Description tables, are described in this appendix.

_infoblk
C.1

The _infoblk table is created whenever the SYSTEM=UNICOS directive is used. This directive is normally found in the default directives file, and _infoblk is normally created for all UNICOS programs. The table contains general information, such as the size of various program sections, time and date of program creation, and version of the loader. _infoblk is structured as follows:

0 32 63

Word				
0:	vers	////	a	len
1:	name			
2:	cksum			
3:	date			
4:	time			
5:	pid			
6:	pvr			
7:	osvr			
8:	udt			
9:	fill			
10:	tbase			dbase
11:	tlen			dlen
12:	blen			zlen
13:	cdatalen			lmlen
14:	amlen			mbase
15:	hinit			hinc

<table>
<tr><th>Word</th><th>0</th><th>32</th><th>63</th></tr>
<tr><td>16:</td><td>sinit</td><td colspan="2">sinc</td></tr>
<tr><td>17:</td><td>usxf</td><td colspan="2">usxl</td></tr>
<tr><td>18:</td><td>mtptr</td><td colspan="2">cmptr</td></tr>
<tr><td>19:</td><td colspan="3">/ / / / / / / / / / /</td></tr>
<tr><td>20:</td><td>sgptr</td><td colspan="2">////</td></tr>
<tr><td>21:</td><td>taskstk</td><td colspan="2">taskincr</td></tr>
<tr><td>22:</td><td colspan="3">user1</td></tr>
<tr><td>23:</td><td colspan="3">user2</td></tr>
</table>

Table 6. `_infoblk` description

Field	Word	Bits	Description
`vers`	0	0–6	`infoblk` table version (currently equals 1).
`a`	0	31	`fill` Address Generation flag (used by the system startup routine to insert address in filled words).
`len`	0	32–63	Number of words in `_infoblk` (currently 24).
`name`	1	0–63	ASCII `_infoblk` table name (“infoblk”). Null-terminated.
`cksum`	2	0–63	Check sum of `_infoblk` contents.
`date`	3	0–63	Date of program creation in ASCII *mm/dd/yy* format.
`time`	4	0–63	Time of program creation in ASCII *hh:mm:ss* format.
`pid`	5	0–63	ASCII name of loader that created program. Null-terminated if name is less than 8 characters.
`pvr`	6	0–63	ASCII version of loader that created program. Null-terminated if name is less than 8 characters.
`osvr`	7	0–63	ASCII operating system active when program was created. Null-terminated if name is less than 8 characters.
`udt`	8	0–63	Date and time of program creation in UNICOS time-stamp format.

Table 6. _infoblk description
(continued)

Field	Word	Bits	Description
fill	9	0–63	Value used by system startup routine to fill uninitialized areas of memory.
tbase	10	0–31	Base address of program text address space.
dbase	10	32–63	Base address of program data address space.
tlen	11	0–31	Number of words in text section.
dlen	11	32–63	Number of words in initialized data section.
blen	12	0–31	Number of words in uninitialized data section.
zlen	12	32–63	Number of words in zeroset data section.
cdatalen	13	0–31	Number of words in initialized data section prior to compressed data expansion.
amlen	14	0–31	Number of words of auxiliary memory used.
mbase	14	32–63	Base address of managed memory area.
hinit	15	0–31	Initial size of program heap.
hinc	15	32–63	Heap expansion increment value.
sinit	16	0–31	Initial size of program stack.
sinc	16	32–63	Stack expansion increment value.
usxf	17	0–31	First address of $USXMSG jump table.
usxl	17	32–63	Last address of $USXMSG jump table.
mtptr	18	0–31	Address of machine targeting information block.
cmptr	18	32–63	Address of first entry in data compression entry list.
sgptr	20	0–31	Address of $SEGRES segmentation information block.
taskstk	21	0–31	Initial size of slave task stack.
taskincr	21	32–63	Task stack expansion increment.

Table 6. _infoblk description
(continued)

Field	Word	Bits	Description
user1	22	0–63	Reserved for users.
user2	23	0–63	Reserved for users.

The contents of the _infoblk table may be accessed from a C language routine by including the following statements:

```
#include <infoblk.h>
extern struct infoblk _infoblk;
```

Segmentation tables
C.2

The loader builds several tables into each segmented program. These tables are used by the segmentation routines included in the program to manage the segments in memory. The $SEGRES table contains general segmentation information, including the addresses of the other segmentation tables. The Segment Description table (SDT) contains one entry for each segment in the program. Each SDT entry describes the size, location, and residency status of each segment. The Segment Linkage table (SLT) contains one entry for each intercepted subroutine call that may result in loading a new segment. Each SLT entry describes the target segment and address needed to complete the subroutine reference.

$SEGRES *table*

A.1.1

The $SEGRES table can be accessed through the common block /$SEGRES/. The other tables must be located through the addresses contained in $SEGRES. The $SEGRES format is as follows:

<table>
<tr><td>Word</td><td>0</td><td></td><td></td><td></td><td>32</td><td>63</td></tr>
<tr><td>0:</td><td colspan="6">length</td></tr>
<tr><td>1:</td><td>d</td><td>c</td><td>s</td><td colspan="2"></td><td>vers</td></tr>
<tr><td>2:</td><td colspan="6">xfer</td></tr>
<tr><td>3:</td><td colspan="6">fill</td></tr>
<tr><td>4:</td><td colspan="4">numslt</td><td colspan="2">bslt</td></tr>
<tr><td>5:</td><td colspan="4">numsdt</td><td colspan="2">bsdt</td></tr>
<tr><td>6:</td><td colspan="4">numjtbl</td><td colspan="2">bjtbl</td></tr>
</table>

Table 7. $SEGRES description

Field	Word	Bits	Description
length	0	0–63	Number of words in $SEGRES table.
d	1	0–0	Flag indicating segmentation debug mode.
c	1	1–1	Flag indicating that segments should be copied to a scratch file.
s	1	2–2	Flag indicating that split segment mode is active.
vers	1	58–63	$SEGRES table version (currently equals 2).
xfer	2	0–63	Address of user main entry point.
fill	3	0–63	Fill value used to preset the uninitialized data section of each segment.
numslt	4	0–31	Number of entries in Segment Linkage table.
bslt	4	32–63	Base address of Segment Linkage table.
numsdt	5	0–31	Number of entries in Segment Description table.
bslt	5	32–63	Base address of Segment Description table.

Table 7. $SEGRES description
(continued)

Field	Word	Bits	Description
numjtbl	6	0–31	Number of entries in interception jump table.
bjtbl	6	32–63	Base address of interception jump table.

Segment Linkage table C.2.1

The Segment Linkage table (SLT) is included in every segmented program. The SLT describes the inter-segment linkages in the program. The Segment Linkage table entry format is as follows:

0	32	63
sdtp	iaddr	

Table 8. SLT description

Field	Word	Bits	Description
sdtp	0	0–31	Address of SDT entry for target segment.
iaddr	0	32–63	Parcel address of target routine.

Segment Description table

C.2.2

The Segment Description table is included in every segmented program. It describes each segment included in the program. The Segment Description table entry format is as follows:

<table>
<tr><th>Word</th><th colspan="3">0</th><th colspan="2">32 63</th></tr>
<tr><td>0:</td><td colspan="5">name</td></tr>
<tr><td>1:</td><td>r</td><td>s</td><td></td><td>level</td><td>acount</td></tr>
<tr><td>2:</td><td colspan="3">succp</td><td colspan="2">predp</td></tr>
<tr><td>3:</td><td colspan="3">tlen</td><td colspan="2">tla</td></tr>
<tr><td>4:</td><td colspan="3">dlen</td><td colspan="2">dla</td></tr>
<tr><td>5:</td><td colspan="3">zlen</td><td colspan="2">blen</td></tr>
<tr><td>6:</td><td colspan="5">/ / / / / / / /</td></tr>
<tr><td>7:</td><td colspan="5">tpos</td></tr>
</table>

Table 9. SDT description

Field	Word	Bits	Description
`name`	0	0–63	ASCII name of segment. Null-terminated if name is less than 8 characters.
`r`	1	0–0	Flag indicating memory residency status of segment.
`s`	1	1–1	Flag indicating segment contents should be written to scratch file before overwriting with another segment.
`level`	1	32–47	Level of segment within segment tree.
`acount`	1	48–63	Number of active calls to routines within segment.
`succp`	2	0–31	SDT entry address of memory-resident successor segment.
`predp`	2	32–63	SDT entry address of predecessor segment.
`tlen`	3	0–31	Number of words in segment text section.
`tla`	3	32–63	Base address of segment text section.
`dlen`	4	0–31	Number of words in segment data section.
`dla`	4	32–63	Base address of segment data section.
`blen`	5	0–31	Number of words in segment uninitialized data section.
`zlen`	5	32–63	Number of words in segment zeroset data section.
`tpos`	7	0–63	Byte position within file of segment contents.

Glossary

absolute binary module

A binary module that the linkage editor has bound. All relative addresses within the bound object modules have been resolved. Also, all external and entry points in these modules have been resolved satisfactorily. This module is considered executable. The name for this module comes from COS where it was referenced as `$ABS`.

barrier

In macrotasking, a mechanism to synchronize tasks. Encountering a barrier causes a task to wait until all tasks have reached the barrier.

bin file

Files specified in `BIN` directives, which are specified as `segldr`(1) command-line option-arguments. By convention, `bin` files should be the portion of your program that you have written. See also *object module*.

block

(1) The smallest allocation unit in a file system; a group of contiguous characters recorded on and read from magnetic tape as a unit. Blocks are separated by record gaps. A block and a physical record are synonymous on magnetic tape. Usually, a block is the size of one physical disk sector. (2) A logical term denoting an arbitrary amount of data; generally a synonym for a 4096-byte hardware sector. See also *sector*. (3) A structure defined by each language processor that represents a contiguous area of memory. Blocks can be local to the defining object module (local blocks) or shared between modules (common blocks). A block can contain instructions, data, or both.

branch segment

Any segment in a segment program that is not the root segment. Branch segments are brought into memory when required, and they may be overwritten by other branch segments.

BSS

The part of a program containing uninitialized data. Space for the area is allocated at execution time.

BSSZ

A BSS area that is initialized to zero.

CAL

Cray Assembly Language

CDBX

An interactive, symbolic debugger that can be used to perform source-level debugging while executing programs running under UNICOS.

common block

A block of memory that will be shared by more than one object module. (1) A Fortran data area that contains data that is accessible to multiple parts of a program. `COMMON` is a type of scope declaration in Fortran that makes variables accessible to multiple parts of a program. More than one program module can specify data for a common block, but if a conflict occurs, information from later programs is loaded on top of previously loaded information. A program may declare 0 to 125 common blocks, which can be either labeled or blank. (2) The C language `global data` items generate both a common block and an entry point.

data loading

The process by which a loader inserts data into object module blocks. Occurs explicitly in response to program statements, such as the Fortran `DATA` statement or C language data initialization operation. Implicit data loading of locations within a subprogram code block can also occur if the compiler or assembler so dictates.

DEX

Distributed EXpression table. A DEX contains many expressions that are evaluated at load time. These expressions are used for many purposes. A prominent use is relocation logic.

distributed mode

In PVM message passing, distributed mode handles communications between a Cray MPP system and a Cray PVP host system.

entry point

A location in a program or routine at which execution begins. A routine may have several entry points, each serving a different purpose. Linkage between program modules is performed when the linkage editor binds the external references of one group of modules to the entry points of another module. See also *absolute binary module*, *object module*, and *loader*.

events

Events record the state of a program's execution (for instance, whether or not it has accessed data yet) and communicate that state to other tasks.

executable program The result of the load process. The executable program is a memory image built from the submitted object files and libraries that can be loaded into memory and executed. The default file name for the executable program is `a.out`.

external reference A reference to an entry point defined outside the referencing module. Fortran `CALL` statements and function calls generate external references. The CAL `EXT` pseudo instruction indicates an external reference. C procedure calls and `extern` statements generate external references.

floating The process by which the loader assigns movable blocks to segments.

force-loading The inclusion of a module that has no callers (for example, force-loading is performed on `BLOCKDATA` modules). The `FORCE` directive enables the force-loading of all uncalled entry points.

Global Symbol table The Global Symbol table is appended to the executable program, and contains information describing the modules, local blocks, common blocks, and entry points included in the program.

heap A section of memory within the user job area that provides a capability for dynamic allocation. See “`HEAP` directive,” page 87, or see the heap memory management routines in the *Application Programmer’s Library Reference Manual*, publication SR–2165.

include To make an object module encountered in an object file or library a part of the executable program.

initial transfer address The entry point at which your program begins execution.

library A collection of functions, or routines, that are functionally related, are called from within programs, and perform commonly used tasks. They are not operating system functions. Library functions let you use code that is already written (you do not have to reinvent wheels), make programs less complicated, and make changing programs easier. The loader includes any module in the library in the executable program only if one of the entry points in the module satisfies an external reference from another module included in the executable program. A library usually is built by a library maintenance tool, such as `bld`(1) or `ar`(1). The file name typically ends with `.a`, and the library is sometimes referred to as a `.a` file.

loader Generic term for the system software product that loads a compiled or assembled program into memory and prepares it for execution.

magic number A number UNICOS uses to identify the type of a file.

module (1) A hardware module is the basic building block of Cray Research systems; modules are made of cold plates and printed circuit boards, and fit into the mainframe chassis. (2) A software module is the basic building block of the IOS-E operating system. (3) A Fortran 90 program module is a program (or function) that contains or accesses definitions to be accessed by other program units.

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 that precedes all callers.

object module The executable binary program that SEGDLR produces.

ordered duplicate selection A method of selecting one of several duplicated entry points found in libraries. SEGLDR locates the first module that references the duplicated entry point and then looks for a definition of the symbol in succeeding modules. The first definition found in a succeeding module is the one used. If SEGLDR finds no succeeding definition, the first definition encountered anywhere is used.

partition

(1) A contiguous set of blocks on a logical device that holds a file system. A partition of a logical device corresponds to a slice on a physical device. In file allocation, partitions permit the distribution of files across the physical devices underlying the logical device on which a file system is mounted. (2) A whole or partial disk unit that consists of an arbitrary number of consecutive tracks on a physical disk device.

primary entry point

An entry point specified by the Fortran or Pascal `PROGRAM` statement, the CAL `START` pseudo-op, or the C main function; it serves as the default transfer address for the program. The first primary entry point encountered is the default transfer address.

PVM

Parallel virtual machine. The message-passing model used by the Cray MPP system. It supports message passing between PEs working on the same application on the Cray MPP system, between the Cray PVP system and the Cray MPP system, and among other combinations of systems (including workstations).

relocatable binary module

A binary module that cannot be executed because absolute machine addresses have not yet been set by the loader/linker; addresses are still only relative to others in the module and therefore, they can be relocated to anywhere in hardware memory.

root segment

The segment that occupies the root node of the segment tree; always resides in memory during program execution.

SDT

Segment Description table. The table is constructed by the loader and is included in every segmented program. It describes each segment included in the program.

sector

A unit of disk storage space equal to 4096 bytes (a physical disk area that can store 512 Cray words). It is the smallest unit of transfer to or from a disk drive. The term block is often used rather than sector when discussing the concept at a high level. However, when disk storage space is meant, the term sector is used. See also *block*.

segment

(1) A single node in the tree structure of a segmented program. (2) A 512-word (minimally) piece of the channel buffer that is allocated by a system's `getseg` code, at the request of the MUXIOP; used for system service requests such as central memory peek or poke executed from the OWS-E. (3) A part of a TCP data stream sent from TCP on one host to TCP on another host. Segments include control fields that identify the segment's location in the data stream and a checksum to validate the received data.

Segment Description table

See *SDT.*

Segment Linkage table

See *SLT.*

SLT

Segment Linkage table. The table is constructed by the loader, and is included in every segmented program. The SLT describes the inter-segment linkages in the program.

special purpose program

A program that will not run under control of the UNICOS operating system. Examples of special-purpose programs include the operating system kernel, or stand-alone diagnostics programs.

stack

(1) A data structure providing a dynamic, sequential data list that can be accessed from one end or the other; a last-in, first-out (push down, pop up) stack is accessed from just one end. (2) A dynamic area of memory used to hold information temporarily; a push/pop method of adding and retrieving information is used.

static memory

Memory that is not on the stack or not in the heap.

transfer entry point

The primary entry point that will receive control from the system initialization routine when the program begins execution.

tree trimming

The process by which SEGLDR eliminates modules that are not referenced in the executable program.

unsatisfied external reference

An external reference for which no entry point of that name can be found in any of the object modules scanned by the loader.

Index

#pragma directive, 104
$SEGCALL, 85
$SEGRES, 85
  description, 145
_infoblk table, description, 141
_loaded, and soft externals, 106

A

ABS directive, overview, 35
Absolute binary module, definition, 149
ADA, 1
ADDBSS directive, 90
ALIGN
  directive, overview, 47
  pseudo-op, 47
Allocating Central Memory
  Cray PVP system, 99, 100
  default allocation scheme, 98
  definitions of terms, 96
  overview, 95
  segmented programs, overview, 99
  shared-text allocation scheme, 98
  TEXT, DATA, BSS allocation scheme, 98
Assigning modules to segments, 67
Assignment, block & program duplication
  nonsegmented programs, 75
  overview, 75
  segmented programs, 76
Automatic duplicaiton of movable blocks using FLOAT directive, 80

B

Barrier, definition, 149
bin and lib files, exceptions and differences, 24
BIN directive
  example, 125
  with DUPORDER directive, 32
  with FORCE directive, 30
  with NODEFLIB directive, 28
BIN directive (global)
  example, 26
  overview, 26
BIN directive (segment)
  example, 70
  overview, 69
Bin file, definition, 149
Block
  assignment
    and program duplication, 75
    definition, 75
    nonsegmented, 76
  definition, 149
  movable, handling using FLOAT directive, 80
BLOCKDATA subprograms, 25, 30
Branch segment, definition, 55, 56, 149
BSS sections, and shared-text memory allocation, 98

C

C, 1
CAL
  definition, 149
  version 2, 1
Calling tree, precautions for design, 57
CALLXFER directive, overview, 113
Case conversion
  controlling, 51
  convention, vi
CASE directive, overview, 51
CAUTION message, definition, 137
CDBX, definition, 150
Central Memory, allocation of, 95
CF90, 1
CFT77, 1
CODE attributes, and shared-text memory allocation, 98
Command line
  ld(1), 10
  options, –k, 137
  segldr(1), 4
Command line options
  ld(1), 10
  segldr(1), 4

Command options, 15
Commands, explain(1), 138
COMMENT directive, as segment description directive, 66
COMMENT message, definition, 137
Comments
  example, 37
  overview, 36
  using in SEGLDR, 21
Common block
  allocation, ORDER directive, 97
  assignment
    overview, 76
    segmented block, 79
  definition, 150
  duplication in nonsegmented loads, 76
  names, duplication in segmented loads, 76
  naming, directives for
    COMMONS, 31
    FORCE, 30
    MODULES, 30
  reference map, 135
  use and assignment, 75, 79, 83
    block data routines, 84
    data load restrictions, 83
    duplicate common blocks, 77
    referencing data in common blocks, 84
Common block use and assignment, user-assigned common blocks, 83
COMMONS directive (global)
  and duplicated common blocks, 76
  overview, 31
COMMONS directive (segment)
  in segmented loads, 76
  overview, 68
COMPRESS directive, overview, 52
Control, load map, 35
Controlling entry points and execution, 44–47, 113
Controlling entry points and execution, directives for
  CALLXFER, 113
  EQUIV, 45
  overview, 44, 113
  SET, 46
  START, 113
  UNSAT, 46
  XFER, 44
Controlling error messages, 39
Controlling error messages, directives for
  DUPENTRY, 41
  DUPLOAD, 42
  MLEVEL, 39
  MSGLEVEL, 43
  NODUPMSG, 42
  NOUSXMSG, 43
  overview, 39
  REDEF, 40
  USX, 40
Controlling listings, directives for
  Comments, 36
  ECHO, 36
  MAP, 37
  TITLE, 38
  TRIAL, 35
Controlling loading, 47
Conventions, for loader directives, 21
COPY directive, overview, 73
CPUCHECK directive, overview, 51

D

DATA attributes, and shared-text memory allocation, 98
Data loading, definition, 150
Data loads
  and FORCE directive, 80
  and MODULES directive, 80
  restrictions, 83
    block data routines, 84
    referencing data in common blocks, 84
Debug symbol table, 50
Debugging, symbolic
  CDBX definition, 150
  overview, 50, 114
  SYMBOLS directive, 50
Default, directives file, 17
Default directives files, 19
Default libraries, 25
Default system libraries, 18
DEFDIR directive
  example, 111
  overview, 109
DEFHEAP directive, overview, 117
DEFLIB directive
  example, 29
  overview, 29
DEFSTACK directive, 118
DEX, definition, 150
Differences between segldr and ld, 17
Directive termination, 22

Directives
- ABS, 35
- ADDBSS, 90
- ALIGN, 47
- BIN (global), 26
- BIN (segment), 69
- CALLXFER, 113
- CASE, 51
- comments, 36
- COMMONS (global), 31
- COMMONS (segment), 68
- COMPRESS, 52
- conventions, 21
- COPY, 73
- CPUCHECK, 51
- DEFDIR, 109
- DEFHEAP, 117
- DEFLIB, 29
- DEFSTACK, 118
- DUP, 71
- DUPENTRY, 41
- DUPLOAD, 42
- DUPORDER, 32
- DYNAMIC, 90, 91
- ECHO, 36
- ENDSEG, 66
- EQUIV, 45
- FORCE, 30
- FREEHEAP, 119
- HARDREF, 108
- HEAP, 87
- INCFILE, 114
- INCLUDE, 23
- LBIN, 26
- LIB, 27
- LIBDIR, 111
- LINCLUDE, 23
- LLIB, 28
- LOGFILE, 53
- LOGUSE, 53
- MAP, 37
- MLEVEL, 39
- MODULES (global), 30
- MODULES (segment), 67
- MSGLEVEL, 43
- NODEFLIB, 28
- NODUPMSG, 42
- NOUSXMSG, 43
- OMIT, 34
- ORDER, 97
- ORG, 49
- OUTFORM, 112
- PRESET, 48
- REDEF, 40
- SAVE (global), 72
- SAVE (segment), 70
- SCOMMONS (global), 31
- SCOMMONS (segment), 68
- SEGMENT, 66
- SEGORDER, 74
- SET, 46
- SLT, 72
- SMODULES (global), 30
- SMODULES (segment), 67
- SOFTREF, 108
- specification order, 21
- STACK, 88
- START, 113
- SYMBOLS, 50
- SYSTEM, 114
- TITLE, 38
- TRIAL, 35
- TSTACK, 89
- UNSAT, 46
- USX, 40
- XFER, 44
- ZEROCOM, 116, 117
- ZSYMS, 115

Directives file, default, 17, 19

Directives processing order, 14

Directives, miscellaneous global, 50, 114

Directives, segment
- ENDSEG directive, 66
- segment description, 66
- SEGMENT directive, 66
- tree definition, 65

Directives, zero address, 115

Distributed EXpression table, definition, 150

Distributed mode, definition, 150

DUP directive
- example, 71, 125
- in nonsegmented loads, 75
- in segmented loads, 76, 77
- overview, 71

DUPENTRY directive
- default setting, 18
- keywords, 41
- overview, 41

Duplicate names
  entry point, 75
  module, 75
Duplicated
  common blocks and COMMONS directive, 76
  entry point errors, controlling, 32
Duplication
  definition, 75
  in nonsegmented loads, 75
    common blocks, 76
    entry point names, 75
    module names, 75
  in segmented loads, 76
    common blocks, 77
    entry point names, 77
    module names, 76
  program and block assignment, 75
  using DUP, 71
  using MODULES, 67
DUPLOAD directive, overview, 42
DUPORDER directive
  default setting, 18
  example, 33
  overview, 32
Dynamic common block
  allocation, 90
  and heap memory, 87
DYNAMIC directive
  example, 91, 124
  overview, 90
  with the heap, 92
Dynamic memory management, 87

E

ECHO directive
  as segment description directive, 66
  example, 128
  overview, 36
ENDSEG directive
  example, 67, 124, 125, 128
  overview, 66
ENDTREE directive
  example, 66, 121, 122, 124, 125, 128
  overview, 65
Entry point
  control and execution, 44, 113
  definition, 75, 150
  duplicate names, 75
  duplication
    error message control, 41, 42
    figure, 78
    in nonsegmented loads, 75
    in segmented loads, 77
    processing order, 32
  names, duplication in segmented loads, 75
  value assignment, 46
Entry point references
  testing with _loaded, 106
  testing with flag words, 107
Entry points, to external functions, 103
Environment variable processing, 12, 17
Environment variables
  LDDIR, 12
  LPP, 13
  MSG_FORMAT, 13
  NLSPATH, 13
  SEGDIR, 13
  TARGET, 14
  TMPDIR, 14
EQUIV directive
  example, 45
  overview, 45
Error, messages, 137
  descriptions, 138
  format, 139
  load–time, 137
  run–time, 137
Error messages
  controlling, directives for
    DUPENTRY, 41
    DUPLOAD, 42
    MLEVEL, 39
    MSGLEVEL, 43
    NODUPMSG, 42
    NOUSXMSG, 43
    REDEF, 40
    USX, 40
  overview, 39
  printing according to severity, 39
Events, definition, 150
Examples
  basic loader invocation, 121
  BIN directive, 26, 125
  BIN directive (segment), 70
  block maps, 131
  Comments, 37

common block reference map, 135
DEFDIR directive, 111
DEFLIB directive, 29
DUP directive, 71, 125
DUPORDER directive, 33
DYNAMIC directive, 91, 124
ECHO directive, 128
ENDSEG directive, 67, 124, 125, 128
ENDTREE directive, 66, 121, 122, 124, 125, 128
entry point cross-reference map, 134
entry point duplication, 78
EQUIV directive, 45
HEAP directive, 128
INCLUDE directive, 23
LIB directive, 27
LIBDIR directive, 111
LINCLUDE directive, 23
LLIB directive, 28
MAP directive, 128
map output, 130
MODULES directive, 31, 68, 124, 125, 129
MSGLEVEL directive, 43
NODEFLIB directive, 29
OMIT directive, 34
SAVE directive, 73
SEGMENT directive, 67, 124, 125, 128
segmented load with duplicated modules, 125
TREE directive, 66, 121, 122, 124, 125, 128
tree structure, 121, 122
figure, 123, 126
tree structure with expandable common block, 122
UNSAT directive, 46
Executable program, definition, 151
Executable program control, 34–35, 112
explain(1) command, 138
ext directive, 104
External functions, references to, 103
External reference, definition, 151
External symbols, 104

F

FATAL message, definition, 138
File attribute directive, OUTFORM, 112
File naming directives
ABS, 35
BIN, global, 26
LBIN, 26
LIB, 27
LLIB, 28
NODEFLIB, 28
Flag word usage, 107
Flag words, testing entry point references with, 107
FLOAT directive, 80
using automatic duplication, 80
using floating, 80
Floating, 80
definition, 151
FORCE directive
and data loads, 84
default setting, 18
overview, 30
Force-loading, definition, 151
Force-loading, definition, 30
Fortran program examples
acquiring space from the heap, 92
basic, 121
comprehensive
block maps, 131
common block reference map, 135
entry point cross-reference map, 134
Fortran program, 126
loader directives, 128
loader map output, 130
source code, 126
using dynamic common blocks, 93
FREEHEAP directive, 119

G

General directives, 21
Global directives
for segmentation, 72
miscellaneous
CASE, 51
COMPRESS, 52
CPUCHECK, 51
FLOAT, 80
INCFILE, 114
LOGFILE, 53
LOGUSE, 53
ORDER, 97
overview, 50, 114
SYMBOLS, 50
SYSTEM, 114
ZSYMS, 115

segmentation
COPY, 73
overview, 72
SAVE, 72
SEGORDER, 74
SLT, 72
Global heap memory, managing, 87, 117
Global segmentation directives, 72
COPY, 73
SAVE, 72
SEGORDER, 74
SLT, 72
Global symbol table, definition, 151

H

Hard references, converting to, 107
HARDREF directive, overview, 108
Header
creating, 38
example, 38
HEAP directive
example, 128
overview, 87
with DEFSTACK directive, 118
with DYNAMIC directive, 91
with STACK directive, 88
Heap memory
and Dynamic Common Block, 90
and the stack, 88, 118
global managing, 87, 117
Heap memory management directives
ADDBSS, 90
DEFHEAP, 117
DEFSTACK, 118
FREEHEAP, 119
HEAP, 87
STACK, 88
TSTACK, 89

I

INCFILE directive, overview, 114
Include, definition, 151
INCLUDE directive
example, 23
overview, 23
Including directives files, 22–23
Including object modules, 24–34
Initial transfer address, definition, 151
Initialization & alignment, program, 47
Initializing data areas, 47
Introduction, 1
Invoking SEGLDR, 3
ld command line, 10
segldr command line, 4

L

Languages supported, 1
LBIN directive, overview, 26
ld(1), command line, 10
LDDIR environment variable, 12
lib and bin files, exceptions and differences, 24
LIB directive
and BIN files, 27
example, 27
overview, 27
with DEFLIB directive, 29
with DUPORDER directive, 32
with FORCE directive, 30
with NODEFLIB directive, 28
LIBDIR directive
example, 111
overview, 111
Library, definition, 152
LINCLUDE directive
example, 23
overview, 23
LLIB directive
example, 28
overview, 28
Load map control, 35–38
Load–time messages, 137
Loader
definition, 152
invocation, 3
Loader directives, 15
conventions, 21
specification order, 21
Loader messages, 137
Loader resident routine ($SEGRES), 85
Loader-created tables, overview, 141
Loading
control of, 37

program segments, 1
LOGFILE directive, overview, 53
LOGUSE directive, overview, 53
LPP environment variable, 13

M

Machine characteristic checking, controlling, 51
Magic number, definition, 152
Managing global heap memory directives
ADDBSS, 90
DEFHEAP, 117
DEFSTACK, 118
FREEHEAP, 119
HEAP, 87
overview, 87, 117
STACK, 88
TSTACK, 89
Map control, load, 37
MAP directive
example, 128
overview, 37
Map output
example, 130
generation, 37
Mass storage, writing segment state to, 70, 72
Memory, overlays, 1
Message
descriptions, 138
format, 139
log file, format, 54
numbers, 138
Messages
controlling error, 39
directives for
DUPENTRY, 41
DUPLOAD, 42
MLEVEL, 39
MSGLEVEL, 43
NODUPMSG, 42
NOUSXMSG, 43
REDEF, 40
USX, 40
Miscellaneous global directives, 50, 114
MLEVEL directive, overview, 39
Module
assigning to segments, 66
definition, 152
duplication
in nonsegmented loads, 75
in segmented loads, 76
using DUP, 71
using MODULES, 67
using SMODULES, 67
names, duplication in nonsegmented loads, 75
names, duplication in segmented loads, 76
naming directives
FORCE, 30
MODULE (segment), 67
MODULES, (global), 30
SMODULE (segment), 67
SMODULES, (global), 30
zero address, 115
MODULES directive
and data loads, 84
example, 124, 125
in segmented loads, 76
MODULES directive (global)
example, 30
overview, 30
MODULES directive (segment)
example, 68
overview, 67
Movable block, definition, 152
MSG_FORMAT, 139
MSG_FORMAT environment variable, 13
MSGLEVEL directive
example, 43
overview, 43

N

Naming files
ABS directive, 35
BIN directive, (global), 26
LBIN directive, (global), 26
LIB directive, 27
LLIB directive, 28
NODEFLIB directive, 28
NLSPATH environment variable, 13
NODEFLIB directive
example, 29
overview, 28
NODUPMSG directive, overview, 42
Nonsegmented program, definition, 1
NOTE message, definition, 137

NOUSXMSG directive, overview, 43

O

Object module
definition, 152
including, 34
OMIT directive
example, 34
overview, 34
ORDER directive, overview, 97
Ordered duplicate selection, definition, 152
ORG directive, overview, 49
OUTFORM directive, overview, 112
Output file attributes, OUTFORM directive, 112

P

Parallel virtual machine, definition, 153
Partition, definition, 153
Pascal, 1
PRESET directive, overview, 48
Primary entry point, definition, 153
Program alignment and initialization, 47–49
directives
ALIGN, 47
ORG, 49
PRESET, 48
overview, 47
Program duplication & block assignment, 75
Program execution, segmented, 85
Program segmentation, introduction, 55
Program segments, loading, 1
Pseudo instructions and shared-text memory allocation, 98
PVM, definition, 153

R

REDEF directive, overview, 40
Relocatable binary module, definition, 153
Restrictions, data load, 83
Root segment, definition, 153
Root segment, definition, 55
Routines, block data, 84
Run-time messages, 137

S

SAVE directive
example, 73
with COPY directive, 73
SAVE directive (global), overview, 72
SAVE directive (segment), overview, 70
SCOMMONS directive (global), overview, 31
SCOMMONS directive (segment), overview, 68
Scratch file
controlling position of, 85
execution from, 73
SDT, description, 147
SDT, definition, 153
SECTION pseudo instructions, and shared-text memory allocation, 98
Sector, definition, 153
SEGDIR environment variable, 13
SEGLDR, invocation statements, 3
segldr(1), command line, 4
Segment
convention, 1
definition, 154
definition directives, 65–66
description directives, 66–71
description, termination, 65
linkage table (SLT), 72
naming, 66
predecessor, function, 55
root, definition, 55
subroutine calling between, 59
successor, function, 55
tree
definition directives, 65
design and restrictions, 57
figures, 56, 57, 58, 59, 61, 62
structure, 55
Segment branch, definition, 56
Segment description directives
BIN, 69
COMMONS, 68
DUP, 71
ENDSEG, 66
MODULES, 67
SAVE, 70
SCOMMONS, 68
SEGMENT, 66
SMODULES, 67
ZEROCOM, 116, 117

Segment Description table, definition, 153
Segment Description Table (SDT), description, 147
SEGMENT directive
  and DUP directive, 71
  example, 67, 124, 125, 128
  overview, 66
Segment Linkage Table, definition, 154
Segment Linkage Table (SLT), 72
  description, 146
Segment tree
  definition directives, 65–66
  restrictions, 57
Segment tree concept, 55
Segment tree definition, 55
Segment tree design, 57
Segmentation
  features, 55
  global directives, 72
Segmentation tables
  $SEGRES, 145
  overview, 144
  SDT, 147
  SLT, 146
Segmented load
  and COMMONS directive, 31
  and SCOMMONS directive, 31
  program, definition, 1
  with duplicated modules, 125
    example, 125
Segmented program, definition, 1
SEGORDER directive, overview, 74
SET directive, overview, 46
SHARED allocation scheme, overview, 98
Shared text program, 97
  creation, 97
  generation, 6
Shared-text allocation scheme
  advantages, 99
  disadvantages, 99
SLT, description, 146
SLT, definition, 154
SLT directive, overview, 72
SMODULES directive (global), overview, 30
SMODULES directive (segment), overview, 67
Soft externals
  how to convert to, 108
  how to convert to hard references, 107
  how to declare, 104–105
  how to link, 105
  overview, 103
  references, 103
  usage, 106–107
  usage (figure), 104
Soft references, 103
Soft references, converting to, 108
SOFTREF directive, overview, 108
Special purpose program, definition, 154
Stack and heap memory, 87, 117
STACK directive, 88
START directive
  command line equivalent for, 10, 16
  overview, 113
Subroutine
  call overhead, overview, 86
  calling between segments, 59
  illegal references, diagram, 61
SUMMARY message, definition, 138
Supported languages, 1
Symbolic debugging directives
  overview, 50, 114
  SYMBOLS, 50
SYMBOLS directive, overview, 50
SYSTEM directive, overview, 114
System libraries, default, 18

T

Tables
  $SEGRES, 145
  _infoblk, 141
  Segment Description Table (SDT), 147
  Segment Linkage Table (SLT), 146
TARGET environment variable, 14
Termination, of segment description, 65
TEXT, DATA, BSS, allocation scheme for memory allocation, 98
TITLE directive
  as segment description directive, 66
  example, 38
  overview, 38
TMPDIR environment variable, 14
Transfer entry point, definition, 154
TREE directive, example, 66, 121, 122, 124, 125, 128
TREE segment definition directive
  example, 66
  overview, 65

Tree structure
- basic example, 121
- figure, 126
- figures, 121, 122
- with expandable common block example, 123

Tree structure, example (figure), 123
Tree trimming, definition, 154
TRIAL directive, overview, 35
TSTACK directive, 89

U

UNICOS
- environment variable processing, 12
- ld(1) command line, 10
- segldr(1) command line, 4

UNSAT directive
- example, 46
- overview, 46

Unsatisfied external reference, definition, 154
Unsatisfied external references, 105
USX directive
- default setting, 18
- overview, 40

W

WARNING message, definition, 138

X

XFER directive
- command line equivalent for, 4, 15
- overview, 44

Z

Zero address, description directives, 115–117
ZEROCOM directive, overview, 116, 117
ZSYMS directive, overview, 115

Reader's Comment Form

Segment Loader (SEGLDR) and `ld` *Reference Manual* SR–0066 9.0

Your reactions to this manual will help us provide you with better documentation. Please take a moment to complete the following items, and use the blank space for additional comments.

List the operating systems and programming languages you have used and the years of experience with each.

Your experience with Cray Research computer systems: _____0-1 year _____1-5 year _____5+years

How did you use this manual: _____in a class _____as a tutorial or introduction _____as a procedural guide _____as a reference _____for troubleshooting _____other

Please rate this manual on the following criteria:

	Excellent			Poor
Accuracy	4	3	2	1
Appropriateness (correct technical level)	4	3	2	1
Accessibility (ease of finding information)	4	3	2	1
Physical qualities (binding, printing, illustrations)	4	3	2	1
Terminology (correct, consistent, and clear)	4	3	2	1
Number of examples	4	3	2	1
Quality of examples	4	3	2	1
Index	4	3	2	1

Please use the space below for your comments about this manual. Please include general comments about the usefulness of this manual. If you have discovered inaccuracies or omissions, please specify the number of the page on which the problem occurred.

Name ______________________

Title ______________________

Company ______________________

Telephone ______________________

Today's date ______________________

Address ______________________

City ______________________

State/Country ______________________

Zip code ______________________

Electronic mail address ______________________

Cut along this line

Fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 6184 ST. PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY
RESEARCH, INC.

ATTN: Software Publications Group
655 LONE OAK DR BLDG F
EAGAN MN 55121-9957

Fold

Reader's Comment Form

Segment Loader (SEGLDR) and `ld` *Reference Manual* SR–0066 9.0

Your reactions to this manual will help us provide you with better documentation. Please take a moment to complete the following items, and use the blank space for additional comments.

List the operating systems and programming languages you have used and the years of experience with each.

Your experience with Cray Research computer systems: _____0-1 year _____1-5 year _____5+years

How did you use this manual: _____in a class _____as a tutorial or introduction _____as a procedural guide _____as a reference _____for troubleshooting _____other

Please rate this manual on the following criteria:

	Excellent			Poor
Accuracy	4	3	2	1
Appropriateness (correct technical level)	4	3	2	1
Accessibility (ease of finding information)	4	3	2	1
Physical qualities (binding, printing, illustrations)	4	3	2	1
Terminology (correct, consistent, and clear)	4	3	2	1
Number of examples	4	3	2	1
Quality of examples	4	3	2	1
Index	4	3	2	1

Please use the space below for your comments about this manual. Please include general comments about the usefulness of this manual. If you have discovered inaccuracies or omissions, please specify the number of the page on which the problem occurred.

Name ____________________

Title ____________________

Company ____________________

Telephone ____________________

Today's date ____________________

Address ____________________

City ____________________

State/Country ____________________

Zip code ____________________

Electronic mail address ____________________

Cut along this line

Fold

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 6184 ST. PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY
RESEARCH, INC.

ATTN: Software Publications Group
655 LONE OAK DR BLDG F
EAGAN MN 55121-9957

Fold