

i960[®] Processor Software Utilities User's Guide

Order Number: 485277-007

Revision	Revision History	Date
-001	Original Issue.	12/92
-002	Minor corrections.	09/93
-003	Revised for CTOOLS960 R4.5 and GNU/960 R2.4.	05/94
-004	Revised for R5.0.	02/96
-005	Revised for R5.1.	01/97
-006	Revised for R6.0	12/97
-007	Revised for R6.5	12/98

In the United States, additional copies of this manual or other Intel literature may be obtained by writing:

Intel Corporation
Literature Sales
P.O. Box 5937
Denver, CO 80217-9808

Or you can call the following toll-free number:

1-800-548-4725

In locations outside the United States, obtain additional copies of Intel documentation by contacting your local Intel sales office.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's Software License Agreement, or in the case of software delivered to the government, in accordance with the software license agreement as defined in FAR 52.227-7013.

Copyright © 1988, 1989, 1990, 1991, 1992, 1993, 1994, 1995 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

Intel Corporation retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

* Other brands and names are the property of their respective owners.



printed on
recycled paper

Copyright © 1992-1994, 1996, 1997, 1998. Intel Corporation. All rights reserved.

Contents

Chapter 1 Overview

Software Utilities and Related Tools	1-1
Compatibility.....	1-3
Compatibility Invocation Names	1-3
DOS No Longer Supported as a Host.....	1-4
Invocation Command-line	1-4
Invocation Names	1-4
Options, Arguments and Modifiers	1-5
File System Dependencies.....	1-5

Chapter 2 Archiver (arc960, gar960)

Invocation	2-1
Option and Modifier Arguments.....	2-3
Specifying the Object Module Format.....	2-4
Temporary Directory	2-5
Option and Modifier Reference	2-5

Chapter 3 COFF/ELF/b.out Converter (cof960/objcopy)

Invocation	3-2
Output File Specification	3-3

Chapter 4 COFF to IEEE-695 Converter (cvt960)

Invocation	4-1
Limitations	4-3
Position-independent Code, Data, and Symbols	4-3
Archives and Relocatable Objects.....	4-3
Unreferenced Types.....	4-3

	Global Uninitialized Symbols	4-3
	Compilation/Assembly Information	4-4
	COFF Line Numbers	4-4
	COFF Symbol Translation	4-5
	IEEE-695 Built-in Types	4-5
	IEEE-695 Converter Warning Messages.....	4-8
Chapter 5	Coverage Analyzer (gcov960)	
	Invocation	5-1
	Example 1	5-6
	Example 2.....	5-6
	Example 3.....	5-7
Chapter 6	Dumper/Disassembler (dmp960, gdmp960)	
	Invocation	6-1
	Dumping Absolute Symbols	6-5
	Examples	6-6
	Archive Support	6-11
	Displaying Archive Structure Information.....	6-12
	Dumping the Contents of Archive Members	6-14
Chapter 7	Linker (lnk960, gld960)	
	Overview	7-1
	Understanding Memory Blocks and Sections.....	7-2
	ELF/DWARF Sections	7-4
	Named BSS Sections	7-4
	Working with Linker Directive Files.....	7-4
	Linker Invocation.....	7-6
	Specifying Object Files	7-11
	Specifying Libraries	7-12
	Specifying Linker-directive Files	7-12
	Naming the Output File.....	7-13
	Incremental Linking	7-13

Object Module Format Compatibilities	7-14
Link-time Optimization	7-16
Using calljx with the i960® RP Processor	7-19
Binding Profile Counters to Non-Standard Sections.....	7-19
Environment Variables.....	7-20
Library Naming Conventions and Search Order.....	7-21
Library Search Order When i960 RP Architecture Is Selected	7-24
Linker Options Reference	7-24
Chapter 8 Macro Processor (mpp960)	
mpp960 Message Prefixes	8-1
Invoking mpp960	8-2
Lexical and Syntactic Conventions.....	8-5
Names.....	8-5
Quoted Strings	8-5
Other Tokens	8-6
Comments.....	8-6
How to Invoke Macros	8-6
Macro Invocation.....	8-6
Macro Arguments.....	8-7
Quoting Macro Arguments	8-7
Macro Expansion	8-8
How to Define New Macros.....	8-8
Defining a Macro	8-8
Arguments to Macros	8-9
Special Arguments to Macros.....	8-10
Deleting a Macro	8-11
Renaming Macros	8-12
Temporarily Redefining Macros.....	8-13
Indirect Call of Macros.....	8-13
Indirect Call of Built-ins.....	8-14

Conditionals, Loops and Recursion.....	8-14
Testing Macro Definitions	8-14
Comparing Strings.....	8-15
Loops and Recursion.....	8-16
How to Debug Macros and Input.....	8-16
Displaying Macro Definitions	8-16
Tracing Macro Calls	8-17
Controlling Debugging Output	8-18
Saving Debugging Output	8-19
Input Control	8-19
Deleting Whitespace in Input.....	8-19
Changing the Quote Characters	8-20
Changing Comment Delimiters.....	8-21
Saving Input	8-22
File Inclusion	8-22
Including Named Files	8-22
Searching for Include Files	8-23
Diverting and Undiverting Output	8-23
Diverting Output	8-24
Undiverting Output	8-25
Diversion Numbers.....	8-26
Discarding Diverted Text	8-26
Macros for Text Handling	8-26
Calculating Length Of Strings.....	8-26
Searching For Substrings	8-27
Searching for Regular Expressions	8-27
Extracting Substrings	8-28
Translating Characters	8-28
Substituting Text by Regular Expression	8-28
Formatted Output	8-29
Macros for Doing Arithmetic.....	8-30

Decrement and Increment Operators	8-30
Evaluating Integer Expressions	8-30
Running Host Commands	8-31
Executing Simple Commands	8-31
Reading the Output of UNIX Commands.....	8-32
Exit Codes.....	8-32
Making Names for Temporary Files	8-32
Printing Error Messages	8-33
Exiting from mpp960	8-34
Compatibility with Other Macro Processors	8-34
Extensions in mpp960	8-34
Facilities in UNIX System V m4 not in mpp960	8-36
Chapter 9 Munger (gmung960)	
Chapter 10 Name Lister (gnm960, nam960)	
Chapter 11 ROM Image Builder (grom960)	
Invocation	11-1
Using grom960	11-3
Creating Binary Images	11-4
Converting the Image to Hex Files.....	11-4
Example 1	11-5
Example 2	11-5
Example 3	11-5
Chapter 12 ROM Image Builder (rom960)	
Rommer Invocation.....	12-3
Directive Files	12-3
Directive Reference	12-6
Chapter 13 Section-size Printer (gsize960, siz960)	
Invocation	13-1
Chapter 14 Statistical Profiler (ghist960)	

Overview	14-1
How Statistical Profiling Works.....	14-2
Parameters that Effect Profiling	14-4
Bucket Size (__buck_size).....	14-6
Timer Frequency (__timer_freq)	14-7
Profiling region (__prof_start and __prof_end).....	14-7
Resources Requirements.....	14-10
Library Initialization	14-11
MON960	14-11
IxWorks	14-11
Invocation	14-12
Using ghist960	14-13
Description of File Formats Emitted by the Library	14-15
Binary Data Format.....	14-16
ASCII Data Format.....	14-16
Example Usage of ghist960	14-17
Chapter 15 Stripper (gstrip960, str960)	
Chapter 16 Assembly Language Converter (xlate960)	
Invocation	16-2
Command-line Invocation.....	16-2
Invocation Through the Assembler	16-3
Invocation Examples	16-4
Output File Format.....	16-4
User Interaction.....	16-5
Translation Errors	16-5
Translation Warnings	16-5
Known Limitations.....	16-6
Appendix A Linker Command Language	
Introduction	A-1
Expressions and Operators.....	A-2

Linker Directives Reference	A-4
MEMORY: Configuring Memory Regions	A-5
Default Linker Allocation.....	A-7
SECTIONS: Defining Output Sections.....	A-8
FORCE_COMMON_ALLOCATION: Allocating Space for Common Symbols.....	A-25
DEFINED: Finding Symbols	A-25
ADDR, ALIGN, NEXT: Finding Addresses.....	A-25
SIZEOF: Finding Section Sizes	A-27
STARTUP: Specifying a Startup File	A-27
ENTRY: Defining the Entry Point.....	A-28
PRE_HLL(): Specifying Libraries to be Processed Before the High-level Libraries.....	A-28
HLL: Specifying High-level Libraries.....	A-29
SYSLIB: Specifying Low-level Libraries.....	A-31
[NO]FLOAT: Supporting Floating-point Operations	A-31
SEARCH_DIR: Extending the Search Path	A-32
INCLUDE: Including Additional Directive Files.....	A-32
TARGET: Using the Search Path for Directive Files.....	A-33
CHECKSUM: Preparing for the Bus Confidence Test ..	A-33
OUTPUT: Naming the Output File.....	A-34
Linker Directive Files.....	A-35

Appendix B Finding Information in Object Files

Using the Common Object File Library: COFL.....	B-1
Extracting File Header Information	B-2
Function Reference.....	B-3

Appendix C Common Object File Format (COFF) and Common Archive File Format (CAFF)

Characteristics of COFF	C-1
Definitions and Conventions	C-2
Sections	C-2

Physical and Virtual Address	C-3
File Header	C-3
File Header Declaration	C-4
File Header Flags	C-5
File Type Numbers	C-6
Execution File Header Declaration	C-7
Section Headers	C-7
Section Header Declaration	C-8
Section Header Flags	C-9
Sections	C-11
Relocation	C-11
Relocation Entry Declaration	C-12
Direct Relocation	C-13
IP-relative Relocation	C-13
Line Number Entry	C-14
Symbol Table	C-15
Symbol Table Entries	C-16
Structure for Symbol Table Entries	C-17
Symbols and Inner Blocks .bb/.eb	C-18
Symbols and Functions .bf/.ef, .target	C-20
Special Symbols	C-20
Symbol Name	C-22
Storage Classes	C-23
Storage Classes for Special Symbols	C-25
Call Optimization	C-25
Symbol Value Field	C-26
Section Number Field	C-27
Section Numbers and Storage Classes	C-28
Type Entry	C-30
Type Entries and Storage Classes	C-32
Auxiliary Table Entries	C-33

Filename	C-35
Section	C-35
Tag Name	C-36
End of Structure	C-36
Function	C-36
Array	C-37
End of Blocks and Functions	C-38
Beginning of Blocks and Functions	C-38
Names Related to Structures, Unions, and Enumerations	C-38
Auxiliary Entry Declaration	C-39
String Table	C-41
Access Routines	C-42
Archive Library Format	C-42
The Archive Identification String	C-43
Archive Members	C-44
The Symbol Table	C-47

Appendix D HP/MRI IEEE 695 Object File Format

Terminology	D-2
Nomenclature	D-3
Number Format	D-3
Name Format	D-3
Information Variables	D-4
Line Numbers	D-5
Types	D-5
Object File Components	D-10
Header Part	D-11
AD Extension Part	D-14
Environmental Part	D-15
External Part	D-17
Section Part	D-18

Debug Information Part	D-19
Data Part	D-30
Trailer Part	D-32
HP/MPI IEEE-695 Format Object File Semantics.....	D-32
AD Extension Part and Environment Part.....	D-33
Public/External Part.....	D-33
Section Part.....	D-33
Debug Part	D-34
BB1 Block.....	D-34
BB3 Block.....	D-34
BB4 and BB6 Blocks	D-35
BB5 and BB10 Blocks	D-35
Miscellaneous Records	D-37
Module Miscellaneous Records	D-38
Variable Miscellaneous Records	D-38
Procedure Miscellaneous Records	D-39
General Syntax Rules.....	D-39
Parameters In Miscellaneous Records	D-39
Optional Parameter Fields.....	D-42
Codes for Miscellaneous Records	D-43
Policies for Adding and Modifying Miscellaneous Records.....	D-43
Policies for Generating and Reading Miscellaneous Records.....	D-44

Index

Examples

C-1 File Header Declaration.....	C-5
C-2 Section Header Declaration.....	C-9
C-3 Relocation Entry Declaration	C-12
C-4 Line Number Grouping	C-14
C-5 Line Number Entry Declaration	C-15

C-6	COFF Symbol Table	C-16
C-7	Symbol Table Entry Declaration	C-17
C-8	Nested Blocks	C-19
C-9	Example of a Symbol Table	C-19
C-10	Auxiliary Symbol Table Entry.....	C-39
C-11	Archive Member Headers.....	C-45

Figures

2-1	Archive Member Replace and Update Operations	2-6
7-1	C Program Storage	7-3
12-1	rom960 Rommer Operations.....	12-1
12-2	Data Placement in Memory Image	12-2
12-3	Dimensions of a Memory Image.....	12-20
12-4	split Command Example	12-23
C-1	Object File Format	C-2
C-2	Flag Field Values	C-10
C-3	String Table.....	C-42
C-4	An Archive Library.....	C-43
C-5	An Archive Member.....	C-44
C-6	The Archive Symbol Table	C-48

Tables

1-1	i960 Processor Software Utilities.....	1-2
1-2	Invocation Names for Backwards Compatibility.....	1-3
2-1	Archiver Options	2-2
2-2	Archiver Option Modifiers.....	2-3
2-3	Verbose Modifier Information Display.....	2-19
3-1	cof960 / objcopy Options.....	3-2
4-1	cvt960 Options	4-2
4-2	Mappings Between COFF and IEEE-695 Built-in Types	4-6
5-1	gcov960 Controls	5-2
5-2	gcov960 Options	5-3

6-1	dmp960/gdmp960 Options	6-2
7-1	Linker Options	7-7
7-2	Branch-and-link and System-call Optimization	7-17
7-3	Supported Input/Output Architecture Combinations ..	7-26
9-1	gmung960 Options	9-2
10-1	gnm960/nam960 Options	10-3
10-2	Section Codes	10-4
11-1	grom960 Options	11-2
11-2	Section Specifications	11-3
12-1	rom960 Directives	12-4
13-1	gsize960/siz960 Options	13-2
14-1	Terminology	14-2
14-2	MON960 Library Parameters	14-4
14-3	IxWorks Library Parameters	14-5
14-4	Preferred Methods for Modifying _prof_start and _prof_end	14-8
14-5	Methods of Modifying _prof_start and _prof_end	14-9
14-6	Required Resources	14-11
14-7	ghist options	14-13
14-8	Procedure for Profiling Under MON960 and IxWorks	14-14
14-9	File Formats Generated by the ghist960 Library	14-15
14-10	Binary Data Format	14-16
14-11	ASCII File Format	14-16
14-12	Examples Parameters	14-18
15-1	str960/gstrip960 Options	15-2
16-1	Instructions Translated by xlate960	16-1
16-1	xlate960 Options	16-3
A-1	Order of Precedence for Operators	A-2
A-2	Linker Directives	A-4
A-3	Memory and Section Attributes	A-7

A-4	SECTION Keywords	A-10
A-5	COFF Binary Representation of NOLOAD, DSECT, COPY Sections	A-23
A-6	ELF Binary Representation of NOLOAD, DSECT, COPY Sections	A-24
B-1	Common Object File Library (COFL) Functions	B-1
B-2	Common Object File Interface Macros	B-3
C-1	File Header Contents	C-4
C-2	File Header Flags	C-5
C-3	Architecture Types of File Header Flags	C-6
C-4	Standard Output (a.out) File Header	C-7
C-5	Section Header Contents	C-8
C-6	Section Header Flags	C-10
C-7	Relocation Entry Format	C-11
C-8	Relocation Types	C-12
C-9	Symbol Table Entry Format	C-17
C-10	Special Symbols in the Symbol Table	C-21
C-11	Symbol Name Field	C-23
C-12	Storage Classes	C-23
C-13	Restricted Storage Classes	C-25
C-14	Symbol Value Field	C-26
C-15	Section Number Field	C-28
C-16	Section Number and Storage Class	C-29
C-17	Fundamental Types	C-31
C-18	Derived Types Field Values	C-31
C-19	Type Entries by Storage Class	C-32
C-20	Auxiliary Symbol Table Entries	C-34
C-21	Format for Auxiliary Table Entries	C-35
C-22	Tag Name Entries	C-36
C-23	Table Entries for End of Structure	C-36
C-24	Table Entries for Function	C-37

C-25 Table Entries for Array	C-37
C-26 End of Block and Function Entries	C-38
C-27 Beginning of Block and Function Entries	C-38
C-28 Entries for Structures, Unions, and Enumerations.....	C-39
C-29 Size and Contents of Archive Member Headers.....	C-46
D-1 Initial Bytes of IEEE Elements	D-2
D-2 HP/MRI IEEE-695 Object-file Representation of High-level Types	D-5
D-3 HP/MRI IEEE-695 Object-file Built-in Types.....	D-8
D-4 Processor Names.....	D-12
D-5 Attribute Definitions for the AD Extension Part.....	D-15
D-6 Attribute Definitions for the Environmental Part.....	D-16
D-7 Attribute Definitions for the External Part	D-18
D-8 Summary of Permitted Block Nesting.....	D-22
D-9 Attribute Numbers, Blocks, and Descriptions	D-27
D-10 Miscellaneous Record Codes.....	D-46

This chapter introduces the i960® processor software utilities and their documentation. It also describes the conventions used throughout this manual.

Software Utilities and Related Tools

The i960 processor software utilities are part of a toolset for developing embedded applications for the i960® Sx, Kx, Cx, Jx, Hx, and Rx processors.

This toolset contains a C/C++ compiler, several libraries, an assembler, a debugger, and the utilities described in this manual and in on-line hypertext. For information on all the related documentation, including the tools hypertext, see your *Getting Started with the i960® Processor Development Tools* manual.

Each utility also has a help option that displays a summary of the utility's invocation options.

Table 1-1 lists the software utilities that are described in this manual.

Table 1-1 i960® Processor Software Utilities

Utility	Names	Function	See
archiver	arc960, gar960	creates and maintains libraries and archives.	Ch. 2
converters	cof960 /objcopy, cvt960	reorder bytes as big-endian or little-endian and convert between b.out format, COFF, ELF, and IEEE-695 format.	Ch. 3, 4
coverage analyzer	gcov960	facilitates testing of i960 processor software applications.	Ch. 5
dumper/ disassembler	dmp960, gdmp960	disassembles and displays object and archive file contents.	Ch. 6
linker	gld960, lnk960	combines object files into executable or relocatable files.	Ch. 7
macro processor	mpp960	creates and interprets macros.	Ch. 8
munger	gmung960	modifies text section and/or data section load address(es) in an object file.	Ch. 9
name lister	gnm960, nam960	prints object file and library symbol table information.	Ch. 10
ROM image builders	rom960, grom960	create a memory image file from an object file.	Ch. 11, 12
section-size printer	size960, gsize960	displays section and file sizes of object files and libraries.	Ch. 13
statistical profiler	ghist960	generates information about application's runtime behavior.	Ch. 14
stripper	gstrip960, str960	removes symbolic information from an object file.	Ch. 15
Assembly Language Converter	xlate960	converts assembly language code from 80960 core processors (e.g., i960 Cx, Jx, and Hx processors) to its CORE0 (e.g., 80960Rx) equivalent.	Ch. 16

Compatibility

Code generated by Release 6.5 is fully compatible with code generated with the Release 5.0, 5.1 and 6.0 tools. Further, source programs compiled with Release 5.0 are accepted by Release 6.5 without change. Almost all environment variables and invocation options are unchanged. Object modules generated with Release 5.0, 5.1, 6.0 can be linked with objects created with Release 6.5. However, object modules compiled with Release 5.0 for the i960 Rx processor should be recompiled with Release 6.5 in order to generate objects that are forward compatible with future i960 Rx processors.

The software utilities also accept output from CTOOLS960 Release 3.0 and later, and from GNU/960 Release 1.2 and later.

Compatibility Invocation Names

Table 1-2 lists the invocation names to use for backwards compatibility with GNU/960 Release 1.3 and later, and for backwards compatibility with CTOOLS960 Release 3.5 and later. (Only tools with more than one name are listed in this table.)

In some cases, using the alternate invocation name causes the tool to behave differently. Invocation name details are provided in the chapter for the tool in question.

Table 1-2 Invocation Names for Backwards Compatibility

Utility	GNU/960	CTOOLS960	See
archiver	gar960	arc960	Ch. 2
b.out / COFF / ELF converter	objcopy	cof960	Ch. 3,4
dumper/disassembler	gdmp960	dmp960	Ch. 6
linker	gld960	lnk960	Ch. 7
name lister	gnm960	nam960	Ch. 10
section-size printer	gsize960	size960	Ch. 13
stripper	gstrip960	str960	Ch. 15

DOS No Longer Supported as a Host

As of release 5.1, CTOOLS no longer supports DOS as a host. For PC development, CTOOLS now supports Microsoft* Windows* NT* and 95. These platforms provide a far more robust development environment, and allow PC users to run CTOOLS without the PharLap* software required by previous versions.

Invocation Command-line

The utility programs and tools described in this manual are command-line driven. This means that to use these tools, you must either:

- type invocation commands in at your host system's command prompt;
- use command scripts, response files, or batch files that can execute on your host; or
- use a tool such as Microsoft `nmake`, Opus `make`, or UNIX* `make`.

There are two similar styles of invocation command-line, resembling other Windows and UNIX program invocation commands. The primary difference between the Windows and UNIX style command-lines is that case (upper and lower) is significant in UNIX paths, directories, and filenames, whereas Windows does not recognize case. The other difference between Windows and UNIX style command-lines involves punctuation of pathnames and invocation options, as described below.

Invocation Names

The command-line for each utility consists of a program name (e.g., `lnk960` or `mpp960`), options (optional), and filenames identifying tool input and/or output. Some tools have more than one invocation name, for compatibility with earlier versions of the tools. (See *Compatibility* on page 1-1.)

Options, Arguments and Modifiers

Invocation command options must be preceded with a special character that identifies them as options. On Windows, this can be either the hyphen character (-) or the slash (/) character. On UNIX, this is the hyphen character. Case is significant in command options, arguments and modifiers, unless the argument is a Windows pathname element. Some invocation options require or accept arguments, and some invocation options have optional modifiers. Some of the tools can be used without options, modifiers or filenames, depending on the tool's default settings. Refer to the chapter describing the desired software tool for information on its invocation options.

File System Dependencies

You must observe the conventions and restrictions of your host environment's file system. Since the example commands in this manual are from the UNIX environment, they work in Windows only if you use the backslash (\) character in pathnames, rather than the slash (/) character.

Archiver (*arc960*, *gar960*)

Use the archiver to create and maintain archive files of:

- ASCII and COFF files (in text or mixed-format archives)
- COFF files (in COFF libraries)
- b.out-format files (in b.out libraries)
- ELF-format files (in ELF libraries)

For information on linking with libraries, see the linker chapter.



NOTE. *To ensure correct formatting, delete and recreate b.out-format libraries created with a GNU/960 archiver before Release 1.3.*

Invocation

Invoke the archiver as:

```
{arc960}
{gar960} -option [-modifier...] archive [name...]
```

arc960 invokes the archiver, providing backwards compatibility with CTOOLS960 R3.5 or later.

gar960 invokes the archiver, providing backwards compatibility with the GNU/960 R1.2 or later.

option is one or more options listed in Table 2-1.

modifier is one or more modifiers listed in Table 2-2.

archive is the archive filename. Unless you specify a complete pathname, the archive must be in the current directory.

names is one or more member or external filenames. Note that when you do not specify a filename or use the -F option, the archiver creates a COFF archive.

Table 2-1 Archiver Options

Option	Effect	See Page
d	deletes members from the archive.	2-10
F	specifies the desired object module format for the an empty library. Object formats are b.out, COFF, and ELF.	2-11
h	displays help information.	2-11
m	moves members to the end of the archive or to the specified position.	2-13
p	prints members in the archive.	2-14
r	replaces existing members or adds new members.	2-15
s	creates or updates library symbol table.	2-16
t	prints member information or the archive table of contents.	2-17
u	updates existing members by the modification dates, or adds new members.	2-17
V	reports the archiver version and continues processing.	2-20
v960	reports the archiver version and stops processing.	2-20
x	extracts members to files without modifying the archive.	2-21
z	suppresses time stamp in archive header.	2-22

Table 2-2 Archiver Option Modifiers

Modifier	Used With	Effect	See Page
a <i>pname</i>	r, u, or m	adds or positions members after the <i>pname</i> member.	2-7
b <i>pname</i> i <i>pname</i>	r, u, or m	adds or positions members before the <i>pname</i> member.	2-8
c	r or u	suppresses the archive-creation message.	2-9
l	any option	uses the current working directory for temporary files.	2-12
o	x	uses the member last-modified date as the extracted file-creation date.	2-14
s	any option or none	creates or updates the library symbol table.	2-16
u	r	updates members only when the member last-modified date is older than the file last-modified date.	2-17
v	any option	reports the archiver progress.	2-19

Option and Modifier Arguments

Invoke the archiver with an option and one or more modifiers. Some modifiers can operate without the option (see Table 2-2).

Some modifiers require arguments. The archiver interprets any string following such a modifier as the argument. Omitting an argument at the end of the command line causes an error. For example:

```
arc960 -rb xy
No archive name specified
```

Specifying the Object Module Format

By default the archiver creates libraries in COFF format. However, the default is overridden when you:

- specify a different format with the `-F` option. (See page 2-11)
- specify a object file on the command line that is not in COFF format.

For example, to create an empty archive in COFF format enter the command:

```
gar960 -u libname.a
```

To create an empty library in ELF format, use the `-F` option:

```
gar960 -Felf -u libname.a
```

However, when you instruct the archiver to add a non-COFF object file to a new archive, the archiver uses the object file's format. For example, if you enter the command:

```
gar960 -u libname.a elf_omf_file.o
```

the archiver creates a new library in ELF format. When you use the `-F` option to specify a library format and try to add an object file in a different format, the archiver uses the format of the object file. For example:

```
gar960 -Felf -u libname.a coff_omf_file.o
```

creates a new library in COFF format.

Once a library is created, all object files within it must be of the same format. For example, trying to add an ELF format object file to a COFF format archive produces an error message.

Temporary Directory

The archiver creates and deletes temporary files. You can choose the temporary working directory:

- Specify the local (`l`) modifier with a directory argument in the archiver invocation.
- Define the `TMPDIR` environment variable. When you do not use the `l` modifier, the archiver uses the directory specified in `TMPDIR`.
- Without either `l` or `TMPDIR`, the archiver uses the directory specified in `P_tmpdir`, defined in the `stdio.h` standard C header file.

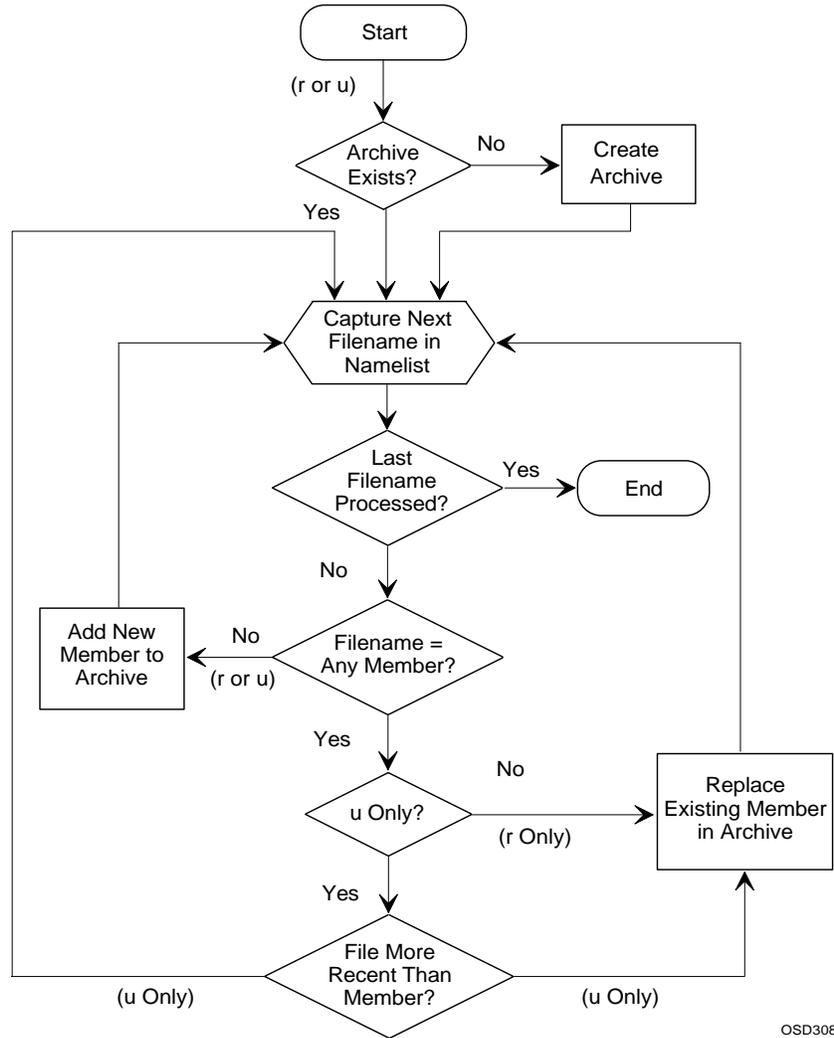
Option and Modifier Reference

You can provide the archiver with the names of external files that you would like added to the archive, or you can tell the archiver the names of any archive members that you would like extracted into external files.

Use the options and modifiers for the following operations:

- To create an archive or to modify the archive members, use `r` or `u` (see Figure 2-1).
- To modify an archive without modifying the members or external files, use `m` or `s`.
- To delete the members, use `d`.
- To modify the external files, use `x`.
- To print information about the archive and its members without modifying the archive contents, use `p` or `t`.
- To display the archiver operation, use `v`.
- To display the archiver version information, use `V` or `v960`.

Figure 2-1 Archive Member Replace and Update Operations



OSD308

This section describes the options and modifiers alphabetically.

a: After

*Modifier: Positions
members after a
specified member*

a pname

pname

is the name of an archive member used as a positional reference.

Discussion

By default, the archiver places new members at the end of the archive. To place new archive members immediately after an existing member, use *a* with replace (*r*) and update (*u*). To reposition existing members, use *a* with move (*m*).

If you specify more than one position modifier, the last one takes effect.

Examples

1. The following example places `file2.o` immediately after `file1.o` in `libx.a`:

```
arc960 -ma file1.o libx.a file2.o
```

2. The following example replaces existing members in `libo.a` with corresponding files from the current working directory, positioning any new members immediately after `file1.o`:

```
arc960 -ra file1.o libo.a *.o
```

Related Topics

Before	Move	Update
Insert	Replace	

b: Before

i: Insert

*Modifier: Positions
members before a
specified member.*

```
b pname
i pname
```

pname

is the name of the archive member used as a positional reference.

Discussion

By default, the archiver places new members at the end of the archive. To place new archive members immediately before an existing member, use **b** or **i** with **replace** (**r**) and **update** (**u**). To reposition existing members, use **b** or **i** with **move** (**m**).

If you specify more than one position modifier, the last one takes effect.

Examples

1. The following commands both place `file2.o` immediately before `file1.o` in `libx.a`:

```
arc960 -mb file1.o libx.a file2.o
arc960 -mi file1.o libx.a file2.o
```

2. The following example replaces existing members in `libo.a` with corresponding files from the current working directory, positioning any new members immediately before `file1.o`:

```
arc960 -rb file1.o libo.a *.o
```

Related Topics

After	Move	Update
Insert	Replace	

c: Create

Modifier: Suppresses the archive creation message

c

Discussion

To suppress the archive creation message, specify `c`.

Using the replace (`r`) or update (`u`) option with a nonexistent archive filename creates a new archive. The archiver displays a message such as the following on `stdout`:

```
arc960: creating archive
```

`archive` is the name of the new archive that you specified.

Example

Assuming that `libx.a` does not already exist, the following example creates an empty `libx.a` archive without displaying an archive creation message:

```
arc960 -rc libx.a
```

d: Delete

Option: Deletes archive members

d

Discussion

To delete all members specified on the command-line, use `d`. In release 6.0, the archiver no longer strips the symbol table information from an archive file when an element is deleted.

Example

The following example deletes `file1.o` from `libx.a`:

```
arc960 -d libx.a file1.o
```

F: Library Format

Option: Specifies the OMF for an empty library

F{elf | coff | bout}

Discussion

This option specifies the desired object module format for the empty library. Object formats are b.out, COFF, and ELF.

Example

The following example creates an ELF archive with member libx.c.

```
arc960 r -Felf libx.a libx.c
```

h: Help

Option: Displays help information

h

Discussion

To display help information for the archiver, use the h option.

I: Local

Modifier: Places temporary files in the current directory

1

Discussion

To put temporary files in the current directory, use 1. The archiver selects a temporary working directory as described in “Specifying the Object Module Format

By default the archiver creates libraries in COFF format. However, the default is overridden when you:

- specify a different format with the `-F` option. (See page 2-10)
- specify a object file on the command line that is not in COFF format.

For example, to create an empty archive in COFF format enter the command:

```
gar960 -u libname.a
```

To create an empty library in ELF format, use the `-F` option:

```
gar960 -Felf -u libname.a
```

However, when you instruct the archiver to add a non-COFF object file to a new archive, the archiver uses the object file’s format. For example, if you enter the command:

```
gar960 -u libname.a elf_omf_file.o
```

the archiver creates a new library in ELF format. When you use the `-F` option to specify a library format and try to add an object file in a different format, the archiver uses the format of the object file. For example:

```
gar960 -Felf -u libname.a coff_omf_file.o
```

creates a new library in COFF format.

Once a library is created, all object files within must be of the same format. For example, trying to add an ELF format object file to a COFF format archive produces an error message.

Temporary Directory” on page 2-4.

Example

The following example replaces `file1.o` and `file2.o` in `libx.a`, using the current directory for temporary files:

```
arc960 -rl libx.a file1.o file2.o
```

m: Move

*Option: Repositions
archive members*

m

Discussion

To reposition members within the archive, use `m`. To move members relative to another member, specify the after (`a`), before (`b`), or insert (`i`) modifier. Omitting the position modifier moves the members to the end of the archive.

Example

The following example places `file1.o` at the end of `libx.a`:

```
arc960 -m libx.a file1.o
```

o: Output Date

Modifier: Extracts a member using the last-modified date

o

Discussion

When extracting a member to a file, the archiver uses the current time and date as the file-creation time stamp. To use the member last-modification time and date, specify o with x (extract).

Example

The following example extracts the `file.o` member, creating the `file.o` external file with the time and date the `file.o` member was last modified:

```
arc960 -xo libx.a file1.o
```

p: Print

Option: Prints archive members

p

Discussion

To display the member contents on `stdout`, use p. To display all the members, specify no member name.

To display each member name before its contents, specify the v (verbose) modifier.

Example

The following example displays the `source1.s` member of `lib.src`:

```
arc960 -pv lib.src source1.s
```

r: Replace

Option: Replaces existing archive members and adds new members

`r`

Discussion

To replace or add a member, regardless of the last-modification dates, use `r`. Existing members specified on the command line are replaced. New members specified on the command line are added. To replace existing members from all filenames in the current directory, specify no member names.

To designate a location for new members relative to existing members, use the after (`a`), before (`b`), or insert (`i`) position modifier.

To create an archive, specify `r` for an archive that does not exist. The following creation message appears:

```
arc960: creating archive
```

archive is the name of the archive created.

You can suppress the creation message with the create (`c`) modifier.

Creating an archive includes creating the new archive symbol table.

Examples

1. The following example adds `file1.o` before `file2.o` in `libx.a`:

```
arc960 -rb file2.o libx.a file1.o
```
2. The following example archives all `.o` files in the current working directory by adding to or replacing members in `libx.a`. The verbose (`v`) option displays messages on the terminal screen, indicating whether members are added (`a -`) or replaced (`r -`).

```
arc960 -rv libx.a *.o  
  
r - hello.o  
r - proto.o  
r - prog_84a.o  
a - str_b.o  
a - str_eg2.o  
a - str_eg3.o  
a - str_eg4.o
```

s: Symbol Table

Option and modifier:
Rebuilds the archive
symbol table

s

Discussion

To rebuild the symbol table, use `s`. You need not specify `s` with the `d`, `r` or `u` option, since the archiver updates the symbol table automatically.

The `s` option has no effect on text archives.

Example

The following example rebuilds the symbol table of `libo.a`:

```
arc960 -s libo.a
```

t: Table of Contents

Option: Lists the member names

```
t [names]
```

Discussion

To list all the members in an archive or to list specified members, use `t`. The *names* argument lists the members for which you want information. To print a table of contents for all members, omit *names*.

The information about specified members appears on `stdout`.

To list the names, permissions, sizes, dates and times of the specified members, specify the verbose (`v`) option. Otherwise, the archiver displays only the member names.

Example

The following example copies the `libs.a` table of contents to `contents.txt` in the current working directory:

```
arc960 -t libs.a > contents.txt
```

u: Update

*Option or Modifier:
Updates archives by*

*comparing the file and
member dates*

u

Discussion

To add or replace members with newer versions, specify u. List specific members to add or replace. Omitting the member list updates only the members with corresponding external filenames in the current directory.

Updating replaces a member only when the date and time stamps on the external file are newer than on the member.



NOTE. *The u option has the same effect both as a separate option and as a modifier of the replace (r) option.*

Examples

1. The following example places all .o files from the current working directory into libo.a. Existing members are updated and new members are added.

```
arc960 -u libo.a *.o
```
2. The following example uses u as a modifier for the replace (r) option, updating file.o only if the external file is more recent than the archived version:

```
arc960 -ru libx.a /newfiles/file.o
```

Related Topic

Replace

v: Verbose

Modifier: Prints the archiver progress information

v

Discussion

For complete information about the archiver operation, specify v. The verbose information is specific to the option you specify, as shown in Table 2-3.

Table 2-3 Verbose Modifier Information Display

Control	Screen Display	Meaning
d (delete)	d - <i>name</i>	removes a member
m (move)	m - <i>name</i>	moves a member within the archive
r (replace) and u (update)	r - <i>name</i> a - <i>name</i>	replaces a member adds a member
x (extract)	x - <i>name</i>	extracts a member into an external file
p (print)		prints the member name and contents
t (toc)		prints the archive table of contents

NOTE: *name* is the name of the processed archive member.

Example

The following example replaces existing members in `libx.a` with the corresponding files in the current directory, placing new members before `file1.o`. As new members are added and existing members replaced, the archiver generates appropriate messages.

```
arc960 -vrb file1.o libx.a *.o
```

On a Windows host, the output is:

```
r - file1.o
a - t.o
a - file2.o
a - file3.o
a - file4.o
```

V, v960: Version

Option: Displays the archiver version number and creation date

$$\left\{ \begin{array}{l} v \\ v960 \end{array} \right\}$$

Discussion

To display a sign-on message during archiving, use `v`. After displaying the message, the archiver continues processing.

To display the message without archiving, use `v960`. You need not provide any other input. After displaying the message, the archiver stops.

The message includes the version number of the archiver and the date and time the archiver was created.

Example

The following command displays the archiver version information and continues processing:

```
arc960 -v
```

The following is a sample message:

```
Intel 80960 Archiver n.n, Thu Oct 19 15:14:07 EDT 199n
```

x: Extract

*Option: Extracts
archive members to files*

x

Discussion

To copy members to external files in the current directory, use `x`. To extract all members, list none on the command line. The contents of the copied archive member are not affected.

The archiver overwrites files in the working directory with the same names as the copied members. If no such files exist, the archiver creates them. The extracted files retain the file attributes stored in the archive, including the modification date and time. To retain the time stamp of when the file was recorded in the library, use the `o` option.

Example

The following example extracts all members of `libs.a` into the current directory. The verbose (`v`) modifier displays the name of each member extracted.

```
arc960 -xv libs.a
```

z: Suppress Time Stamp

*Option: Suppresses
time stamp in the
archive header*

z

Discussion

To suppress the time stamp in the archive header, use the z option.

Example

```
arc960 -z libs.a
```

COFF/ELF/b.out Converter

(cof960/objcopy)

3

cof960 and objcopy convert header information in object files to allow greater portability between hosts. Note that cof960/objcopy does not change the data portion of a file. The converter takes an input file and creates an output file with one of these conversions:

big-endian byte order to little-endian or vice-versa.

- COFF to b.out format or vice-versa.
- ELF to b.out or vice-versa.
- ELF to COFF or vice-versa.

Or you can instruct the converter to remove relocation and symbolic information to reduce code size of debugged objects.

Changing the byte order may be necessary for symbolic debugging, and for examination with common object format library (COFL) tools and other utilities. In such cases, COFF object-file byte order may need to match the host system's byte order. For example, certain debuggers require it, as do the tools COFL and cvt960.



NOTE. *The converter does not initialize the space between ELF sections. Although this does not affect the validity of the ELF file, it can produce unexpected differences between otherwise identical files. Use the dumper/disassembler (gdump960/dmp960) with the -m option to determine the location of these gaps in an object file.*

Invocation

Invoke the cof960/objcopy converter as:

```
{ cof960 }
{ objcopy } [-option]... input [output]
```

<i>cof960</i>	invokes the converter for backwards compatibility with CTOOLS960 R3.5 and later.
<i>objcopy</i>	invokes the converter for backwards compatibility with GNU/960 R1.3 and later.
<i>option</i>	is any option listed in Table 3-1.
<i>input</i>	is the input file.
<i>output</i>	names the output file. Without an output filename specification, the converter overwrites the input file.

Table 3-1 **cof960 / objcopy Options**

Option	Effect
b	generates big-endian output, regardless of the input byte order.
c	copies the file rather than moving it.
C	strips CCINFO from output.
Fbout	specifies the output format, regardless of the input format.
Fcoff	• Fbout generates b.out-format output.
Felf	• Fcoff generates COFF output.
	• Felf generates ELF output.
	Omitting F leaves the format unchanged. Changing the file format removes the relocation information and symbol table.
h	generates an output byte order matching the host-system byte order, regardless of the input byte order.
help	displays help information and exits.

continued 

Table 3-1 **cof960 / objcopy Options** (continued)

Option	Effect
J	Compresses the symbol table, merges duplicate tags, and compresses the string table (COFF files only).
l	generates little-endian output, regardless of the input byte order.
p	overwrites the input files with the converted output.
q	suppresses printing status information during conversion.
r	converts little-endian input to big-endian output. Using r with big-endian input causes an error.
S	removes all relocation and symbol-table information. Do not use S when converting a library or other relocatable files, as it renders the files useless.
v	displays progress information during the conversion (converter lists the action taking place).
V	displays (on stdout) the converter version number and the date and time the converter was created, then continues processing.
v960	displays (on stdout) the converter version number and the date and time the converter was created, then stops processing.
x	removes the local symbols generated by the gcc960 compiler.
z	uses Time Zero instead of the current time and date for the COFF output-file time stamp. Time Zero is 4:00, 31 December, 1969.

Output File Specification

You can direct the converted output either to overwrite the input or to produce a different file:

- To put the converted output in another file, preserving the input file, follow the input filename with the output filename in the invocation. For example, the following puts the big-endian conversion of `lfile.o` into `bfile.o`:

```
cof960 -b lfile.o bfile.o
```

3

- To overwrite the input files, specify the `p` option. When converting the file contents in place, you can provide multiple input files for each converter invocation. Separate the input files with spaces. For example, the following converts `file1.o`, `file2.o`, and `file3.o` from little-endian to big-endian:

```
cof960 -bp file1.o file2.o file3.o
```

COFF to IEEE-695 Converter (cvt960)

4

The COFF-to-IEEE converter (cvt960) converts files in Common Object File Format (COFF) to IEEE-695 format. The IEEE-695 format conforms to the *IEEE-695 Object Module Format Specification*, Revision 4.0, Copyright 1987-1989, by Microtec Research Incorporated and Hewlett-Packard. Note that cvt960 cannot translate b.out or ELF format files into IEEE-695 format.

cvt960 requires that the input COFF file provided for translation be in host-endian orientation. Use objcopy/cof960 to translate the input file into host-endian orientation prior to executing cvt960.

Invocation

To convert files in common object file format (COFF) to IEEE-695 format, invoke cvt960 with the following syntax:

```
cvt960 [-option]...
```

cvt960 invokes the converter.

option is one or more of the options listed in Table 4-1.



NOTE. To convert a file named `a.out`, putting the IEEE-695 format output in `a.x`, invoke the converter with no options.

Table 4-1 **cvt960 Options**

Option	Effect
a	converts files for use with the MRI Xray user interface (pre-X263 and earlier versions).
Aarch	specifies an 80960 architecture tag for the IEEE-695 output file. Valid <i>arch</i> values are CORE0, CORE1, CORE2, CORE3, SA, SB, KA, KB, CA, CF, JA, JD, JF, JT, HA, HD, HT, RD, RP, RM, RN and VH. See <i>Getting Started</i> for a description of the new CORE0-CORE3 group architecture options.
c	specifies emitting column zero for line entries rather than one.
h	displays help information.
i <i>input</i>	specifies an input COFF file. The default input filename is a.out.
o <i>output</i>	names the IEEE-695 output file. The default output file is the input filename with the extension replaced by .x. When the input filename has no suffix, the converter appends .x.
s	suppresses the IEEE-695 public and debug parts. The converted file contains no line number, symbol table, or debug information.
V	displays converter version information and continues processing.
v960	displays converter version information and stops processing.
w	suppresses warning messages.
z	writes constant time stamp (Time Zero) and command line to output file. Time Zero is 4:00, 31 December, 1969.

Examples

- The following converts a.out to a.x:


```
cvt960
```
- The following displays the converter version information and translates hpx.in into hpx.x in /e/asm_tests/cvt_960:


```
cvt960 -V -i /e/asm_tests/cvt_960/hpx.in
```
- The following translates a.out into hpx.out, stripping the debug information:


```
cvt960 -s -o /e/asm_tests/cvt_960/hpx.out
```

Limitations

This section describes parts of the COFF, the converter, or the IEEE-695 format that can cause conversion problems.

Position-independent Code, Data, and Symbols

The converter translates position-independent code and data correctly, but position-independent code symbols and position-independent data symbols lose the flags that mark them as position-independent.

Archives and Relocatable Objects

Use the converter only on COFF absolute-executable load modules. The converter does not translate archives or relocatable objects.

Unreferenced Types

The converter does not produce type definitions for high-level types that are not referenced. This omission helps to reduce the size of the IEEE-695 module where the C `#include` mechanism has produced a large number of unreferenced type definitions, such as structure tags.

Global Uninitialized Symbols

With IEEE-695, every symbol is owned by some source module. The structure of the COFF symbol table, however, dictates that symbols for global, uninitialized variables belong to no specific source module. In order to translate COFF global variables, the converter produces a module in `.bss` strictly for symbols that are not accounted for in any other module. This module is named `.global_non_init`. A single-module section is produced for `.global_non_init`, which extends from the lowest-addressed symbol in the module to the end of the `.bss` section.

Compilation/Assembly Information

COFF does not include source file path information, and the compiler and assembler tools before V3.0 do not supply the time of compilation and assembly for source modules. The converter does not supply this information.

COFF Line Numbers

COFF does not provide column information for source coordinates and the converter does not provide that information. Column numbers in the IEEE-695 output module are 0.

The converter translates each COFF line-number record to a IEEE-695 ATN/ASN pair, possibly causing one-to-many mappings in the output module numbers, as in the following examples:

- COFF source-line information provides the same code address for the line of a function block's `{` token and the first executable line of code. Using the code fragment below, for example, the converter puts the line containing `{` and the line containing `int foo = 1` (lines 2 and 4) in the COFF symbol table with the same code address.

```
1:  main()
2:  {
3:
4:      int foo = 1;
```

- COFF source-line information provides multiple addresses for the same source line under some conditions. For example, a `while` loop associates the source line of the `while` statement with the machine address of the branch to the loop-condition test at the end of the loop. The machine code associated with the loop-condition test produces an additional line-number entry with the same row number as the branch. These two line number groups are translated for the `while` loop with the same line number and different addresses.

COFF Symbol Translation

The compiler prefixes most C language names with an underscore (`_`) when creating COFF symbols. When the converter finds a symbol with an initial underscore and a `.file` symbol ending in `.c` or `.i`, it treats the symbol as a C name with the underscore prefix. The converter strips out the initial underscore and places the symbol in the B3 block corresponding to that COFF module. The B3 block describes high-level debug information.

COFF symbols that come from any source modules whose `.file` symbol does not end in `.c` or `.i` are considered assembly language symbols. The converter leaves any initial underscores intact and places the symbols in the B10 block, which describes assembly-level debug information. The symbols are given IEEE-695 assembler-static attributes and built-in types.

IEEE-695 Built-in Types

Table 4-2 lists the translation of COFF symbols of scalar and pointer types to IEEE-695 built-in types. The Valid in Public Part column indicates types produced for symbols in the IEEE-695 public part.

4

Table 4-2 Mappings Between COFF and IEEE-695 Built-in Types

IEEE-695 Code	COFF Concept	Meaning	Valid in Public Part	Notes
0	T_NULL	unknown type	yes	
1	T_VOID	void-return	no	
2	T_CHAR	8-bit signed	yes	
3	T_UCHAR	8-bit unsigned	no	
4	T_SHORT	16-bit signed	no	
5	T_USHORT	16-bit unsigned	yes	
6	T_INT,T_LONG	32-bit signed	no	
7	T_UINT,T_ULONG	32-bit unsigned	yes	
8	n/a	64-bit signed	no	
9	n/a	64-bit unsigned	no	
10	T_FLOAT	32-bit float	yes	
11	T_DOUBLE	64-bit float	yes	
12	T_LNGDBL	extended float	yes	1
13	n/a	128-bit float	no	1
14	n/a	quoted string	no	
15	C_LABEL	code address	yes	
16	n/a	stack push	no	2
17	n/a	stack push	no	2
18	n/a	stack push	no	2
19-24	n/a	alias for above	no	
25	n/a	64-bit BCD float	no	3
26-31		reserved	no	

continued 

Table 4-2 Mappings Between COFF and IEEE-695 Built-in Types (continued)

IEEE-695 Code	COFF Concept	Meaning	Valid in Public Part	Notes
32	DT_PTR	p.unknown type	no	
33	DT_PTR	p.void-return	no	
34	DT_PTR	p. 8-bit signed	no	
35	DT_PTR	p. 8-bit unsigned	no	
36	DT_PTR	p. 16-bit signed	no	
37	DT_PTR	p. 16-bit unsigned	no	
38	DT_PTR	p. 32-bit signed	no	
39	DT_PTR	p. 32-bit unsigned	no	
40	n/a	p. 64-bit signed	no	
41	n/a	p. 64-bit unsigned	no	
42	DT_PTR	p. 32-bit float	no	
43	DT_PTR	p. 64-bit float	no	
44	DT_PTR	p. extended float	no	1
45	n/a	p. 128-bit float	no	1
46	n/a	p. quoted string	no	
47	DT_PTR	p. code address	no	
48	n/a	p. stack push	no	2
49	n/a	p. stack push	no	2
50	n/a	p. stack push	no	2
51-56	n/a	alias for above	no	
57	n/a	64-bit BCD float	no	3
58-255		reserved		

Notes:

- 1 Although ic960 allocates a 128-bit cell with base address to 2^4 for the C "long double" type, the actual datum is 12 bytes long (manipulated by load-and-store triple word instruction). Thus, the ic960 long double type maps to IEEE-695 built-in type #12, even though its memory alignment might suggest built-in type #13.
- 2 These types correspond to stack pushes. The converter does not produce them because the i960 processor family does not have explicit push instructions.
- 3 The i960 processor family has no BCD-float support.

IEEE-695 Converter Warning Messages

The warning messages appear on `stderr`. After a warning, the translation completes, but the output can be unusable.

No public/debug info produced: no `.file` symbols in COFF symbol table

The converter must find at least one `.file` symbol in the COFF symbol table to establish a starting point for translation. When no such symbol is found, the IEEE-695 public and debug parts are not included in the output module.

COFF section id *number* is type COPY;symbol/data conflicts possible in output

The converter found a COPY section in the COFF file. The IEEE-695 format has no direct analog of a COPY section, so the conversion could confuse the user.

COFF section id *number* is type DSECT; symbol conflicts possible in output

The converter issues this message because some linkages may produce symbol tables where two or more symbols point to the same memory. Some emulators cannot handle this, and reject loading such files. The converter gives you this information here to prevent your waiting until an emulator fails to load the files.

Illegal register value (number) at symbol index *number*

The value of the COFF symbol at the indicated index does not represent an i960® processor register. The IEEE-695 translation contains an invalid i960 processor register index.

COFF argument symbol at index *number* is ignored; addressing path too complicated for IEEE-695

The converter cannot process any COFF symbol whose addressing path is more complicated than: `offset (register)`. This limitation only affects C function arguments that are allocated in the caller's argument block.

One or more COFF symbols (*index number*) have invalid tag
index number

The converter encountered COFF symbols of a tag type (*struct*,
union, or *enum*) with no reference to their COFF type information.
The IEEE-695 information for these symbols is not correct.

Coverage Analyzer (*gcov960*)

The *gcov960* test coverage analysis tool performs basic block execution coverage analysis of instrumented programs.

To use *gcov960*, first compile your program with the *gcc960* `fprof` instrumentation option, then execute the program with appropriate input data. (For more information on profiling, see *the i960® Processor Compiler User's Guide*.) Executing your instrumented program causes the compilation system to update the program database and create a profile data file (`default.pf`, by default). You can then use the options described in this chapter to create a variety of reports showing how your program behaves with various inputs.

Invocation

Invoke the coverage analyzer using the syntax:

```
gcov960 [control]... [file [=module, ...]]... [option]...
```

<i>control</i>	is one of the controls listed in Table 5-1.
<i>file</i>	identifies a source file, from the profiled program represented in the program database. Specifying <i>file</i> restricts the operation of <i>gcov960</i> to the file.
<i>module</i>	identifies a module within <i>file</i> .
<i>option</i>	is one or more of the options listed in Table 5-2.

If you supply the optional *file* [=*module*, . . .] input along with the *-x1* option, *gcov960* reads the source and produces an annotated listing of the source along with the coverage data in *file.cov*. In the annotated source, each statement within a basic block is prefixed with the number of times it has been executed. Lines that have not been executed are prefixed with #####.

Note that in this chapter a *basic block* refers to a single entry, single exit code region containing no branching mechanisms. The number of lines marked with ##### may not equal the number of blocks listed in the *gcov960* report.

Table 5-1 **gcov960 Controls**

Control	Argument	Effect
c	p	produces program-level coverage report.
	m	produces module- and program-level coverage report.
	f	produces function-, module-, and program-level coverage report.
	s	produces a source- and module-level coverage report.
f	n	produces the n most frequently executed lines (when n is positive) or least frequently executed (when n is negative).
g	h	produces a call-graph listing of the program. The h argument is optional; it attaches to the report an explanation of how to interpret the call-graph listing.
n	<i>new_profile</i>	compares two profiles. The new profile is compared to the default or the profile specified with the p or iprof option, and just-hit or just-missed lines are reported. Multiple instances of this control and argument are supported; The profiles specified are automatically merged together.
Q	<i>n</i>	ignores hits from functions whose profiles are of increasing accuracy (decreasing levels of interpolation). <i>n</i> is 0..9. The default is Q9, which ignores hits for all functions except those with perfect profiles. Q0 ignores hits for any functions, even those with completely guessed profiles.

Table 5-1 **gcov960 Controls** (continued)

Control	Argument	Effect
r	l	produces annotated source listing.
	h	produces the line numbers within the basic blocks that were hit. You can also use the l option to specify the directories searched.
	m	produces the line numbers within basic blocks that were missed. You can use the l option to specify the directories searched.

Table 5-2 **gcov960 Options**

Option	Argument	Effect
C		calculates the total number of execution counts in a profile.
h		prints help information.
l	<i>search_dir</i>	adds directory to list that <i>gcov960</i> searches for source files.
p or iprof	<i>file</i>	identifies profile to be used. Default is <i>default.pf</i> . Multiple instances of this option and argument are supported. The profiles specified are automatically merged together.
Q	<i>n</i>	Ignores hits except from functions whose profiles are at least <n> accurate. The valid range for <n> is 0-9. The default is -Q9, which ignores profile information for functions that have had source code changes since the profile was collected. -Q0 tells <i>gcov960</i> to use a profile even for functions with profile information that is completely interpolated. A profile's quality gradually drops as changes are made to the code from which it was collected and interpolation is done to make it useable by <i>gcc960</i> , <i>ic960</i> , and <i>gcdm960</i> .
q		suppresses display of version, copyright, profile, and program database used.
t		truncates displayed names to keep them within column widths.
V		prints version and continues.

continued 

Table 5-2 gcov960 Options (continued)

Option	Argument	Effect
v960		prints version and exits.
Z	<i>pdb_dir</i>	identifies program database directory. Default is directory identified by G960PDB (gcc960) or I960PDB (ic960). See the <i>Getting Started</i> manual for more information on environment variables.



NOTE. *The reports produced by gcov960 may give misleading information about functions that are inlined. The reports may indicate that the code of the inlined function has never been executed, or may show execution counts that are unexpectedly low. This occurs because the inlined code fragments are treated as part of the function they are inlined into and not as part of the original function.*

Examples

The following examples assume that you compile and execute the following source file named `compare.c`.

```
/* compare.c */
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
    int n1, n2;
    if (argc != 3)
    {
        printf("Usage: compare n1 n2\n");
        exit(0);
    }
    n1 = atoi(argv[1]);
    n2 = atoi(argv[2]);
```

```
    {
        printf("Use integers larger than zero\n");
        exit(0);
    }
    if (n1 == n2)
        printf("\n%d equals %d\n", n1, n2);
    else
    {
        if (n1 < n2)
            printf("\n%d is less than %d\n", n1, n2);
        else
            printf("\n%d is greater than %d\n", n1, n2);
    }
}
```

To compile `compare.c` use the command:

```
gcc960 -Fcoff -fprof -Z pdb -ACF -Tmcyex compare.c
```

The above command creates the directory `pdb` (if it doesn't exist already) to store the program database information and generates an absolute module named `a.out`, which can be downloaded and executed on a Cyclone board with a i960 CF processor module.

The following command creates the file containing the profile information using `mondb`.

```
mondb -ser a.out 10 10
```

This command creates a file called `default.pf`. Once you have the `default.pf` file you can copy it to a file with a different name such as `default.old`. You can then use `mondb` again to create a new profile with a different set of data, for example:

```
mondb -ser a.out 10 20
```

Example 1

- The gcov960 invocation:

```
gcov960 -c -Z pdb
```

- produces the coverage report shown below.

Intel 80960 Coverage Analyzer n.n.nnn

Copyright (C) 1996 Intel Corporation. All rights reserved.

```

                                Coverage Analysis
                                Program Summary
No.          No. Blocks  No. Block
Blocks       Hits       Misses      Coverage
-----
                13         7         6         53.85%

Program database:  /ffs/a/joe/tmp/pdb
Program profile:  /ffs/a/joe/tmp/default.pf

```

Example 2

- The gcov960 invocation for an instrumented program `compare.c`:

```
gcov960 -rl -Z pdb
```

produces the output in file `compare.cov` shown below. The numbers on the left are the execution count for the basic blocks associated with the statement.

```

                                #include <stdio.h>
                                main(argc, argv)
                                int argc;
                                char *argv[];
1 -> {
                                int n1, n2;
1 ->   if (argc != 3)
                                {
#####->   printf("Usage: compare n1 n2\n");
#####->   exit(0);
                                }

```

```

1 -> n1 = atoi(arv[1]);
1 -> n2 = atoi(arv[2]);
1 1 1 -> if ( (n1 <= 0) || (n2 <= 0))
{
#####-> printf("Use integers larger than zero\n");
#####-> exit(0);
}

1 -> if (n1 == n2)
1 ->     printf("\n%d equals %d\n", n1, n2);
else
{
#####->     if (n1 < n2)
#####->         printf("\n%d is less than %d\n", n1, n2);
else
#####->         printf("\n%d is greater than %d\n", n1, n2);
}
}

No. of Blocks:      13
Blocks Hit:         7
Blocks Missed:     6
Source Coverage:   53.85%
Program database:  /ffs/a/joe/tmp/pdb
Program profile:   /ffs/a/joe/tmp/default.pf

```

Example 3

The gcov960 invocation to compare two profile files (default.pf and default.old) created after running the instrumented program compare.c:

```
gcov960 -rlm -n default.pf -p default.old -Z pdb
```

- produces file compare.cov containing the output shown below.

```

#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
    int n1, n2;

```

5

```
if (argc != 3)
{
    printf("Usage: compare n1 n2\n");
    exit(0);
n1 = atoi(argv[1]);
n2 = atoi(argv[2]);
if ( (n1 <= 0) || (n2 <= 0))
{
    printf("Use integers larger than zero\n");
    exit(0);
}
if (n1 == n2)
    printf("\n%d equals %d\n", n1, n2);
else
{
#####->     if (n1 < n2)
#####->     printf("\n%d is less than %d\n", n1, n2);
        else
            printf("\n%d is greater than %d\n", n1, n2);
    }
}
```

```
Lines Just Missed:  2
No. of Blocks:     13
Blocks Hit:        9
Blocks Missed:     4
Source Coverage:   69.23%
Program database:  /ffs/a/elvis/tmp/pdb
Program profile:   /ffs/a/elvis/tmp/default.old
Other program file: /ffs/a/elvis/tmp/default.pf
```

Dumper/Disassembler (*dmp960*, *gdmp960*)

6

The dumper/disassembler displays object or archive (library) files in COFF, ELF, and b.out formats.

It displays object file contents, including:

- file, section, and COFF optional headers
- line-number entries
- relocation entries
- symbol and string tables
- contents of the sections as assembly language
- contents of the sections as hexadecimal bytes, in little-endian byte order

Invocation

Invoke the dumper as:

```
{ dmp960 }  
{ gdmp960 } [-option]... filenames
```

dmp960 invokes the dumper for backwards compatibility with CTOOLS960 Release 3.5 and later.

gdmp960 invokes the dumper for backwards compatibility with GNU/960 Release 2.0.1 and later.

option an option listed in Table 6-1. Invoking the dumper without any options disassembles the contents of all sections.

filenames one or more filenames, separated by spaces, indicating files to be displayed. You can specify complete pathnames.

Table 6-1 dmp960/gdmp960 Options

Option	Effect
a	disassembles all sections in an object file. Use this to examine the raw DWARF information in a file.
A{ SA SB KA KB CA CF JA JD JF JT HA HD HT RD RP RM RN VH CORE0 CORE1 CORE2 CORE3 ANY}	Specifies architecture for which you are disassembling. This options does not currently affect disassembly.
c	displays the string table.
d	disassembles all sections loaded into target memory. Unless otherwise specified, text sections appear as assembly language, and data sections appear as hexadecimal bytes. When no options are specified when invoking the dumper, -d is assumed as the default option.
D	disassembles sections as hexadecimal bytes, regardless of the section type. The physical address of every fourth word appears at the beginning of each line.
e	applies all command line options to each member of an archive file.
f	displays the file headers.
F <i>function</i>	disassembles the specified COFF or ELF function.

continued ➤

Table 6-1 **dmp960/gdmp960 Options** (continued)

Option	Effect
<i>g[<i>arg</i>]</i> ...	dumps one or more <code>.debug_*</code> sections. The default argument is <code>i</code> (<code>.debug_info</code>). Use the following arguments to specify a different section, or multiple sections: <i>i</i> dump <code>.debug_info</code> (default argument) <i>l</i> dump <code>.debug_line</code> <i>f</i> dump <code>.debug_frame</code> <i>p</i> dump <code>.debug_pubnames</code> <i>a</i> dump <code>.debug_aranges</code> <i>m</i> dump <code>.debug_macinfo</code> <i>A</i> dump all <code>.debug_*</code> sections
<i>h</i>	displays the section headers.
<i>help</i>	displays help information and exits.
<i>i</i>	displays the COFF optional header.
<i>l</i>	displays the line numbers.
<i>m</i>	displays the memory map (section layout) of the file.
<i>n name</i>	displays the requested information for only the specified section.
<i>O filename</i>	displays the requested information for only the specified file within an archive.
<i>o</i>	displays the disassembled addresses in octal. The default is hex.
<i>p</i>	suppresses all the header displays, producing a parseable output.
<i>q</i>	queries the file and displays its type.
<i>r</i>	displays the relocation information.
<i>s</i>	used with <i>d</i> , puts symbols in the disassembly in place of addresses.

continued ➡

Table 6-1 **dmp960/gdmp960 Options** (continued)

Option	Effect
S	Instructs the disassembler to output symbol labels rather than their values for any symbols for which you have specified the absolute address. This option works in conjunction with the -s (lowercase) option, which instructs the disassembler to perform symbolic disassembly.
t	displays the object file symbol-table entries, or an archive's symbol list.
T	displays the disassembled section as assembly language, regardless of the section type.
V	displays the dumper version and creation date, and continues processing.
v960	displays the dumper version and creation date, and stops processing.
x	is ignored unless used with -t. Displays the symbol-table entries as hexadecimal numbers instead of a symbolic translation of debug information. By default, the dumper produces symbolic information.
z	suppresses the translation of zeros into .word directives for text-type disassembly.

Dumping Absolute Symbols

The `-s` option instructs the disassembler to output symbol labels rather than their values for any symbols for which you've specified the absolute address. This option works in conjunction with the `-S` (lowercase) option, which instructs the disassembler to perform symbolic disassembly. For example, with an object file created with the following instructions:

```
.globl    proc1
.set      proc1,0xc
callx    proc1
callx    0xc
addi     proc1,r5,r6
```

If you use the following `gdm960` command line:

```
gdm960 t2.o -s
```

you would see the output:

```
Section '.text':
0: 8600000c          callx    0xc
4: 8600000c          callx    0xc
8: 5931488c          addi     12,r5,r6
```

Notice that in the second line, `proc1` from the source code is converted to `0xc`, the user-specified address for `proc1`.

Adding the `-S` option to the command line instructs the disassembler to display the symbol name instead of its address. For example, this command line:

```
gdm960 t2.o -s -S
```

produces the following output:

```
Section '.text':
0: 8600000c          callx    proc1
4: 8600000c          callx    proc1
8: 5931488c          addi     12,r5,r6
```

Notice that in the both `callx` statements, `proc1` now appears instead of `0xc`. Using the `-S` option causes the disassembler to display the symbol name for all calls to that address.



NOTE. *This option was supported in the rev. 5.0 disassembler as the undocumented `-A` switch. This option has been renamed `-S`.*

Examples

The examples that follow show how you can extract information from object files with `dmp960`. The file `t.c` is a simple C program that is first compiled then assembled.

```
int                arr[12] = { 3, 4 };
static int        index;

main()
{
    int tempt = func( arr[index] );
}

ic960 -S -g t.c
cat t.s

# Command line (ic960): ic960 -S -g t.c # Command line (cc1):
# /ffs/pl/dev/sun4/lib/cc1.960 -ic960 -ffancy-errors -sinfo
# /tmp/ica02371.sin -fno-builtin -quiet -Fcoff -mkb -mic3.0-compatible
# -fsigned-char -wl -bname_tmp /tmp/ica02371.btm -O0 -g -dcmd
# "ic960 -S -g t.c" -dumpbase t /tmp/ica02371.i -o t.s .file "t.c"
ic_name_rules.:
gcc2_compiled.:
__gnu_compiled_c:
.globl    _arr
.data
.align    4
_arr:
.word    3
.word    4
.space   40
.text
.align    4
.def     _main;    .val    _main;    .scl    2;    .type    0x44;
.undef
.globl    _main
# Function 'main'
```

6

Dumper/Disassembler(dmp960, gdm960)

```
# Registers used: g0 g4 g5 g6 g7 g14 fp r4* #
_main:
    .def    .bf;    .val    .;    .scl    101;    .line    5;    .endif
    .ln    1
    addo    16,sp,sp
#Prologue stats:
# Total Frame Size: 16 bytes
# Local Variable Size: 16 bytes
# Register Save Size: 0 regs, 0 bytes #End Prologue#
mov    g14,r4
    .def    _temp;    .val    64;    .scl    1;    .type    0x4;    .en
def
    .ln    2
    ld    _index,g4
    ld    _arr[g4*4],g0
    callj _func
    mov    g0,g4
    mov    g4,g5
    st    g5,64(fp)
    .ln    3
#EPILOGUE:
ret
    .def    .ef;    .val    .;    .scl    101;    .line    3;    .endif
    .def    _main;    .val    .;    .scl    -1;    .endif
    .def    _index;    .val    _index;    .scl    3;    .type    0x4;
.endif
    .bss    _index,4,2
    .def    _arr;    .val    _arr;    .scl    2;    .dim    12;    .size
48;    .type    0x64;    .endif
```

asm960 t.s -o -t.o

6

dmp960 t.o

Section '.text':

```
0: 59084810    addo   16,sp,sp
4: 5c20161e    mov    g14,r4
8: 90a03000    00000060 ld     0x60,g4
10: 90803914    00000030 ld     0x30[g4*4],g0
18: 09ffffe8    call   0x0
1c: 5ca01610    mov    g0,g4
20: 5ca81614    mov    g4,g5
24: 92afe040    st     g5,0x40(fp)
28: 0a000000    ret
```

Section '.data':

```
30: 03000000 04000000 00000000 00000000
40: 00000000 00000000 00000000 00000000
50: 00000000 00000000 00000000 00000000
```

Here is the same example, but with symbolic disassembly enabled. Note how much more closely the disassembly resembles the assembly source code.

dmp960 -s t.o

Section '.text':

_main:

```
0: 59084810    addo   16,sp,sp
4: 5c20161e    mov    g14,r4
8: 90a03000    00000054 ld     _index,g4
10: 90803914    00000024 ld     _arr[g4*4],g0
18: 09ffffe8    call   _func
1c: 9287e040    st     g0,0x40(fp)
20: 0a000000    ret
```

Section '.data':

```
24: 03000000 04000000 00000000 00000000
34: 00000000 00000000 00000000 00000000
44: 00000000 00000000 00000000 00000000
```

Here are the relocations that are passed to the linker.

dmp960 -r t.o

```

*** RELOCATION INFORMATION ***
Section '.text':
Vaddr                               Type   Name
0x0000000c                          RELLONG.bss
0x00000014                          RELLONG.data
0x00000018                          IPRMED_func
0x00000018                          OPTCALL_func
Section '.data':
Section '.bss':

```

And here is part of the COFF symbol table. Note the use of the `-n` option to make the dumper display only symbols from the `.bss` section.

dmp960 t.o -t -n .bss

```

*** SYMBOL TABLE INFORMATION ***
[Index] m1 Name/Offset Value Scnum  Flags Type   Sclass Numaux Name
[Index] a0 Word1 Short1 Short2 Short3 Short4 Short5 Short6 Tv
[Index] a1 Fname
[Index] a2 Tagndx Fsize Lnnoptr Endndx Tvndx
[Index] a3 Scnlen Nreloc Nlinno
[Index] a4 Tagndx Lnno Size Dim[0] Dim[1] Dim[2] Dim[3]
[Index] a5 Identification String Date/Time
[12] m1 _index 0x00000060 3 0x1000 int static 0 _index
[17] m1 .bss 0x00000060 3 0x1200 static 1 .bss
[18] a3 0x00000004 0x0000 0x0000

```

6

Here is part of the b.out symbol table. The s column identifies the section, as in gnm960 output. The o column indicates whether each symbol is ordinary (O), a leaf entry point (L), or a system procedure entry point (S).

```
gcc960 -c -Fbout t.c; gtmp960 -t t.o
```

```
*** BOUT SYMBOL TABLE ***
```

Value	S	O	Symbol Name
0x00000000	t		gcc2_compiled.
0x00000000	t		__gnu_compiled_c
0x00000030	D		_arr
0x00000000	T		_main
0x00000060	b		_index
0x00000000	U		_func
0x00000000	t		.text
0x00000030	d		.data
0x00000060	b		.bss

Here is part of the ELF symbol table. For more information on ELF symbol tables, refer to the *80960 Embedded Application Binary Interface (ABI) Specification*.

```
gcc960 -c -Felf t.c
```

```
gtmp960 -t t.o
```

Index	Value	Size	Binding	Type	Section	Oth Name
0	0x00000000	0	LOCAL	NONE	UNDEFINED	0x0
1	0x00000000	0	LOCAL	SECTION	.text	0x0 .text
2	0x00000000	0	LOCAL	SECTION	.data	0x0 .data
3	0x00000000	0	LOCAL	SECTION	.bss	0x0 .bss
4	0x00000000	0	LOCAL	SECTION	.shstrtab	0x0 .shstrtab
5	0x00000000	0	LOCAL	SECTION	.strtab	0x0 .strtab
6	0x00000000	0	LOCAL	SECTION	.symtab	0x0 .symtab
7	0x00000000	0	LOCAL	SECTION	.rel.text	0x0 .rel.text
8	0x00000000	0	LOCAL	FILE	ABSOLUTE	0x0 t.c
9	0x00000000	0	LOCAL	NONE	.text	0x0 gcc2_compiled.
10	0x00000000	0	LOCAL	NONE	.text	0x0 __gnu_compiled_c
11	0x00000000	0	LOCAL	NONE	.bss	0x0 _index
12	0x00000000	0	GLOBAL	NONE	.data	0x0 _arr
13	0x00000000	0	GLOBAL	NONE	.text	0x0 _main
14	0x00000000	0	GLOBAL	NONE	UNDEFINED	0x0 _func

The last example shows the dumper's display of the COFF section headers:

```
dmp960 t.o -h
```

```
*** SECTION HEADERS ***
```

Name	Paddr	Vaddr	Scnptr	Relptr	Lnnoptr
	Align	Size	Nreloc	Nlnno	Flags
.text	0x00000000	0x00000000	0x00000098	0x000000f4	0x00000124
	0x00000010	0x0000002c	4	4	REG, TEXT
.data	0x00000030	0x00000030	0x000000c4	0x00000000	0x00000000
	0x00000010	0x00000030	0	0	REG, DATA
.bss	0x00000060	0x00000060	0x00000000	0x00000000	0x00000000
	0x00000010	0x00000004	0	0	REG, BSS

Archive Support

With release 5.1 and later, *gdm960* supports dumping of archive files and archive file members. Previous versions of the dumper only worked with object files. Archive support allows you to dump:

- all members of an archive
- one or more object files within an archive
- information on the structure of an archive (e.g., the archive symbol list)

The table below lists the options that allow archive support:

Option	Description
-e ¹	applies all options on the command line (e.g., -r, -f) to each member of an archive.
-m	displays a map of the archive contents. See the first example later in this section.
-O <i>filename</i> ¹	applies all command line options to the named archive member file only.
-p	suppresses headers.
-q	queries the archive file and displays its object module format and host byte order.
-t	displays the archive symbol list.

¹ Indicates a new dumper option.

Displaying Archive Structure Information

The examples that follow show an archive file `lib.a` that contains the object files `a.o`, `b.o`, and `c.o`, in that order.

This first set of examples show how the dumper can display information on the structure of an archive file using the `-q`, `-m`, and `-t` options.

This example demonstrates the behavior of the dumper when querying an archive file for its type. The command:

```
gdmp960 -q lib.a
```

produces the output:

```
File:                lib.a
OMF:                 elf archive
Host Byte Order:    big
Target Byte Order:  unknown
```

In the next example, the dumper maps the internal structure of an archive file. The command:

```
gdmp960 -m lib.a
```

produces the following output:

	HEX	DEC	OCT
	---	---	---
-----+-----	0	0	0
Magic String 0x8 (8)			
-----+-----	8	8	10
Symbol List HDR 0x3c (60)			
-----+-----	44	68	104
Symbol List 0x64 (100)			
-----+-----	a8	168	250
a.o HDR 0x3c (60)			
-----+-----	e4	228	344
a.o 0x2934 (10548)			
-----+-----	2a18	10776	25030
b.o HDR 0x3c (60)			

6

Dumper/Disassembler(*dmp960*, *gdm960*)

```
+-----+ 2a54   10836   25124
|      b.o
|      0x253c (9532)
+-----+ 4f90   20368   47620
|      c.o HDR
|      0x3c (60)
+-----+ 4fcc   20428   47714
|      c.o
|      0x7b74 (31604)
+-----+ cb40   52032   145500
|      END OF FILE
|      0x0 (0)
+-----+ cb40   52032   145500
```

The `-t` option of the dumper permits dumping of the archive symbol list information. For example, the command:

```
gdm960 -t lib.a
```

produces the output:

Name	Offset	Filename
<code>_dwarf_init</code>	168	<code>a.o</code>
<code>_dwarf_tag</code>	10776	<code>b.o</code>
<code>__dw_build_a_die</code>	20368	<code>c.o</code>
<code>__dw_build_tree</code>	20368	<code>c.o</code>
<code>__dw_build_cu_list</code>	20368	<code>c.o</code>

Dumping the Contents of Archive Members

The dumper lets you disassemble or display information about a file within an archive by using the `-e` and `-o` options in combination with other command line switches. In the example below, the `-e` option applies all command line options to each member of an archive:

```
gdmp960 -q -e lib.a
a.o:
File:                a.o
OMF:                 elf
Host Byte Order:    big
Target Byte Order:  little

b.o:
File:                b.o
OMF:                 elf
Host Byte Order:    big
Target Byte Order:  little

c.o:
File:                c.o
OMF:                 elf
Host Byte Order:    big
Target Byte Order:  little
```

The example below shows how the `-o` option lets you apply all command line options to the named object file only. The command:

```
gdm960 -q -Oa.o -Oc.o lib.a
```

produces the following output:

```
a.o:  
File:           a.o  
OMF:           elf  
Host Byte Order:  big  
Target Byte Order: little
```

```
c.o:  
File:           c.o  
OMF:           elf  
Host Byte Order:  big  
Target Byte Order: little
```


Linker (lnk960, gld960)

7

Overview

The linker lets you combine unlinked or partially linked object files and libraries into programs for debugging or execution on any i960® processor. Linking can include:

- configuring a program for the target memory, including the addresses and section combinations
- searching libraries to resolve external references
- adding, preserving, or removing symbolic debugging information
- defining or redefining global symbols
- changing `callj` and `calljx` to branch-and-link or system calls
- patching all relocatable instructions, data and debug information (in ELF/DWARF)

Though you can specify most of the linker options to perform these functions on the command line, most users use a combination of command line options and a linker directive file to provide input to the linker. CTOOLS provides a number of linker directive (`.ld`) files in the `[$G960BASE | $I960BASE]\lib` directory. Typically, users store most commands for allocating memory blocks and configuring memory in the linker directive file. They then use the command line to invoke the linker, specify the object files to be linked, specify the linker directive file, and include any command line options needed to add to or override the settings in the linker directive file.

This chapter focuses on teaching you how to use the linker by providing the following:

- Some basic information about how the linker allocates memory blocks and sections.
- A sample linker directive file that you can edit to match the requirements of your target execution environment.
- Instructions on how to invoke the linker, specify your linker directive file, object files, and other command line options.
- Sample command lines using additional linker features such as `callj/calljx` link time optimization that you may want to use in your software development.
- A complete reference of all linker command line options.

Understanding Memory Blocks and Sections

With the linker, you can specify the portions of i960® processor's address space are available and where within that space the sections of your program will be located. Once you have defined configured memory, all other areas in the address space are left unconfigured, and are unavailable for linking.

A section is the smallest relocatable unit of an object file. The linker supports COFF, b.out, and ELF object module formats. A b.out-format program contains exactly three standard sections:

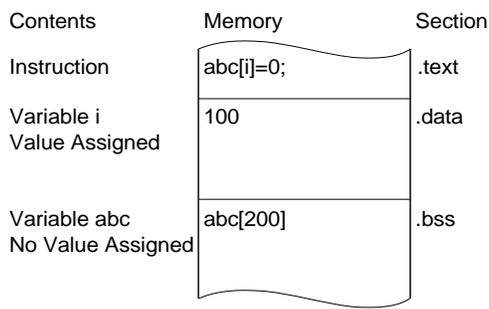
<code>.text</code>	is the standard text-type section, containing instructions and by default starting at address 0.
<code>.data</code>	is the standard data-type section, containing initialized data and by default follows <code>.text</code> .
<code>.bss</code>	(block started by symbol) is a unique section, containing uninitialized data and by default follows <code>.data</code> .

A COFF or ELF program contains at least the three standard sections (containing up to 65535 lines or relocation entries each) COFF section names are restricted to eight or fewer characters; ELF section names can be any length. For information on the COFF file format see Appendix C. For more information on the ELF format, see the *80960 Embedded Application Interface Specification* (Intel order number 631999).

The following example includes two global declarations and an assignment. The translated assignment code is stored in `.text`, the `i` variable in `.data`, and the `abc` variable in `.bss`, as shown in Figure 7-1.

```
int abc[200];      /* in the .bss section, because abc is
                  not initialized in the .data    */
int i=100         /* section because i is initialized */
int f() {        /*
abc[i] = 0;      /* instructions are placed    */
}               /* in the .text section      */
```

Figure 7-1 C Program Storage



OSD312

You can define and link the sections in any order. In your assembly source text and linker directive files, you can create and name additional text-type, data-type and BSS-type sections for COFF and ELF programs. You can also specify section and global-symbol addresses and overlap or suppress some sections.

ELF/DWARF Sections

ELF/DWARF sections are placed in non-allocated sections. The linker concatenates and relocates these sections, but allocates no memory for them. These sections are all allocated to memory address 0. This includes, among other sections, `.debug_info` and `.debug_abbrev`. For more information on the ELF format, see the *80960 Embedded Application Interface Specification* (Intel order number 631999).

Named BSS Sections

Named Block Started by Symbol (BSS) sections are supported by the linker. When the linker detects the symbol `__clear_named_bss_sections` in the linkage, the linker generates code to resolve this symbol. By default, the code is composed of a `ret` instruction only. However, when the linker detects any number of named bss sections in its output, it generates code to clear the sections and places this code in the `__clear_named_bss_sections` code.

Working with Linker Directive Files

The sample linker directive file below is for the Cyclone i960® Cx processor-based evaluation board. By default, the installation program places this file in:

```
[$G960BASE | $I960BASE]\lib\cyex.ld
```

The commands under the `MEMORY` directive define the areas of memory that are available and the type of memory that resides there. This platform has two memory blocks defined: a DRAM region that begins at address `0xA0008000` and is `0x1ff8000` bytes in length, and an SRAM region that

begins at address 0xA0000100 and is 0x300 bytes long. Notice that the DRAM address range allows room for the RAM space required by the on-board monitor.

The commands in SECTIONS specify where the linker places the different program sections in memory.

```
MEMORY
{
    dram :   org = 0xA0008000, len = 0x1ff8000   /* 32M less monitor */
    isram :  org = 0x00000100, len = 0x300
}

SECTIONS
{
    .text :
    {
    } >dram

    .data :
    {
    } >dram

    .bss :
    {
    } >dram

    /* Arithmetic controls location when using floating point library. */
    SFP_AC :
    {
        fpem_CA_AC = .;
    } >isram
}

/* Bounds of heap: */
/* The heap may be placed in a separate memory region, if desired. */
_heap_size = 0x20000;
_heap_base = (_end + 0xf) & ~0xf;
_heap_end = _heap_base + _heap_size - 1;

/* _stackbase marks base of user stack */
/* stack is allocated following the heap. */
/* The stack may be based in a separate memory region, if desired. */
_stackbase = (_heap_end + 0x10) & ~0xf;

STARTUP ("crt960")
HLL ()
SYSLIB ("libmn")
SYSLIB ("libl1")
FLOAT
```

The remaining options specify the floating point and other libraries used. `_heap_size`, `_heap_base`, `_heap_end`, and `_stackbase` are global symbols that define the heap and stack. For more information on the linker directives used in this sample file, see Appendix A.

Linker Invocation

Once you have set up your linker directive file, you are ready to link your object modules. To run the linker, use the syntax:

```
{lnk960}
{gld960} [-options] filenames
```

<code>lnk960</code>	invokes the linker, providing backwards compatibility with CTOOLS960 Release 3.5 or later.
<code>gld960</code>	invokes the linker, providing backwards compatibility with GNU/960 Release 2.2 or later.
<i>options</i>	is one or more of the options listed in Table 7-1.
<i>filenames</i>	is one or more object, library, or linker-directive filenames.

For example, to link the files `file1.o`, `file2.o` using the linker directives in `cycx.ld`, enter the command:

```
lnk960 file1.o file2.o cycx.ld
```

The linker provides many options that let you customize the linking process. Table 7-1 lists these options.

Table 7-1 Linker Options

Name	Option	Effect	Default Action
Architecture	A{ SA SB KA KB CA CF JA JD JF JT HA HD HT RD RP RM RN VH CORE0 CORE1 CORE2 CORE3 }	specifies architecture. The <code>CORE0-3</code> options let you generate code that is compatible with a group of processors. The types of i960® processors supported by each <code>CORE</code> switch are: <code>CORE0</code> Jx, Hx, Rx, VH* <code>CORE1</code> All 80960 architectures, VH* <code>CORE2</code> Jx, Hx, Rx, VH* <code>CORE3</code> Cx, Jx, Hx, VH*	uses the KB libraries.
* Except for Big-Endian mode, which is unsupported in VH.			
Section start address	<code>Bsection addr</code> T {bss data text} <code>addr</code>	assigns section addresses.	locates the sections consecutively in the default sequence.
Startup alternative	C	suppresses any <code>STARTUP</code> directive.	uses the startup routine specified with <code>STARTUP</code> in the directive file.
Circular search	c	searches libraries circularly.	searches libraries once.
Inhibit CAVE compression	D	prevents the linker from compressing CAVE sections.	linker compresses CAVE sections.
Define common symbol space	dc dp	reserves space for common symbols, with the same effect as a <code>FORCE_COMMON_ALLOCATION</code> directive (see Appendix A). Useful only when combined with <code>-r</code> .	does not reserves space for common symbols.

continued 

Table 7-1 Linker Options (continued)

Name	Option	Effect	Default Action
Define symbol	<code>defsym name = expr</code>	defines an absolute symbol.	
Entry point	<code>esymbol</code>	defines the primary entry point.	uses the beginning of .text as the entry point.
Format	<code>Fcoff</code> <code>Fbout</code> <code>Felf</code>	selects COFF, b.out, or ELF as the output format.	defaults to b.out for gld960 and COFF for lnk960.
Fill	<code>fvalue</code>	initializes the gaps within a section.	initializes the memory between sections to 0.
Big-endian target	<code>G</code>	specifies that input files and output files are for a big-endian target. Valid only for COFF or ELF files.	little-endian target files.
Decision maker	<code>gcdm</code>	invokes gcdm960 optimization decision maker.	does not invoke gcdom960.
Sort common symbols	<code>H</code>	sorts common symbols by size.	does not sort common symbols.
Help	<code>h</code>	prints help and terminates.	
Compress	<code>J</code>	merges duplicated tags, and compresses string table. (COFF only).	uses no compression.
Library search path	<code>Ldir</code>	adds a directory to the library and directive-file search path.	uses a search path determined by the output format and invocation name.

continued ➡

Table 7-1 Linker Options (continued)

Name	Option	Effect	Default Action
Library input	<i>labbr</i>	specifies a library-filename abbreviation.	takes no action.
Memory map	<i>m</i>	writes a memory map to stdout.	generates no memory map.
Name map file	<i>Nfile</i>	places memory map in <i>file</i> . Useful only with <i>-m</i> .	sends memory map (if generated) to stdout.
Noinhibit output	<i>n</i>	produces an output file regardless of errors.	for most errors, suppresses the output file.
Optimization inhibit	<i>Ob</i> <i>Os</i>	inhibits the branch-and-link or system-call optimizations.	optimizes the <i>callj</i> and <i>calljx</i> pseudo-instructions.
Output filename	<i>ofilename</i>	names the output file.	produces <i>a.out</i> for COFF output, <i>b.out</i> for b.out-format output, and <i>e.out</i> for ELF format output.
Profiling	<i>P</i>	adds profiling initialization code. This option is useful only when combined with <i>-r</i> .	adds no profiling instrumentation.
Position independence	<i>p{b c d}</i>	marks the output as position-independent (PIC/PID), issuing a warning for non-PIC/PID input. Uses libraries with PIC (<i>-pc</i>), PID (<i>-pd</i>) or both (<i>-pb</i>).	does not mark output as position-independent.
Read symbols only	<i>Rfilename</i>	includes only the symbols from the specified object file.	includes the entire file.
Relocation	<i>r</i>	retains the relocation entries in the output file.	removes the relocation entries.

continued ➡

Table 7-1 Linker Options (continued)

Name	Option	Effect	Default Action
Strip	S,s	-S removes only the debug information from the output file. -s removes all symbolic information from the output file.	retains the symbolic information.
Section start address	<i>Bsection addr</i> T {bss data text} <i>addr</i>	assigns section addresses.	locates the sections consecutively in the default sequence.
Target	<i>Tfilename</i>	uses the full search path for directive filename, with the same effect as a TARGET directive.	searches only the current directory.
Suppress multiple definition symbol warnings	t	suppress warnings of multiple symbol definitions, even if they differ in size.	displays all warnings.
Unresolved symbol	<i>usymbol</i>	puts an unresolved symbol in the symbol table.	suppress warnings of multiple symbol definitions even if they differ in size. No unresolved symbols added to file.
Version	V	displays linker version information.	displays no version information.
Verbose	v	displays linker progress.	displays no progress information.
Version; stop	v960	displays the linker version information and stops linking.	displays no version information and stops linking.
Warnings	W	suppresses warnings except symbol table warnings.	displays all warnings.

continued 

Table 7-1 Linker Options (continued)

Compress	X	removes the local symbols beginning with a dot (.) or L.	retains local symbols.
Compress	x	removes the local symbols.	retains local symbols.
Trace symbol	y <i>sym</i>	traces a symbol, indicating each file where it appears, its type, and whether the file defines or references it.	does not trace symbols.
Program database	Z <i>pdb_dir</i>	specifies location of program database.	linker uses location defined with variable \$G960PDB if defined.
Time stamp suppression	z	puts Time Zero in the output time stamp.	uses current time stamp.

Specifying Object Files

When you specify the input-object and library files on the command line or in the directive files, provide the full object filenames, with the filename extensions. The linker processes the input object and library files in the following order:

1. a file designated with the `STARTUP` directive (the `C` option suppresses the `STARTUP` directive).
2. the object files and libraries listed individually in the invocation, in the order appearing on the command line.
3. the object files and libraries listed individually in the linker-directive files, in the order encountered.
4. the libraries listed with the `HLL` directive.
5. the libraries listed with the `SYSLIB` directive.

Specifying Libraries

To resolve a library reference, list the library after all object files containing the reference. Only required library members are linked. The libraries are opened and searched once (unless you use the `-c` option).

Specify a library file:

- with no option, using the full filename with its extension.
- with the `l` option and a standard library-filename abbreviation, as described in the *Library Naming Conventions and Search Paths* section on page 7-21.
- with the `HLL` directive, for the standard high-level support libraries.
- with the `SYSLIB` directive, for the libraries (such as low-level support libraries) to be linked last.

The following example links the `file1.o` and `file2.o` object files. The user-defined library `libu.a` resolves references in `file1.o`. Any unresolved references in `file2.o` that could be resolved in `libu.a` cause the link to fail.

```
lnk960 file1.o -lu file2.o
```

Using the `c` option to perform a circular library search resolves all symbols.



NOTE. *The linker searches library names and locations according to the new library installation convention. You must upgrade library installations older than release 5.0.*

Specifying Linker-directive Files

A linker-directive file can contain input object and library filenames, directives, options, and other directive filenames.

- To restrict the directive-file search path to the current directory, specify the filename alone on the command line or with `INCLUDE` in another directive file with a restricted search path.
- To use the full search path, use the `T` option or the `TARGET` directive. `INCLUDE` directives in a file included with `T` or `TARGET` also use the full search path.

The following example links the `file1.o` and `file2.o` object files. The object files and the `lnk12.ld` directive file are in the current directory. Instructions and data from `file1.o` are given lower addresses, respectively, than instructions and data from `file2.o`, unless otherwise specified in the directive file.

```
lnk960 file1.o file2.o lnk12.ld
```

For more information on linker directive files, see Appendix A.

Naming the Output File

Unless you specify otherwise, the output file is named:

<code>a.out</code>	for COFF output
<code>b.out</code>	for b.out-format output
<code>e.out</code>	for ELF output

The linker overwrites existing files with the default output names. To preserve an existing `a.out`, `b.out` or `e.out` file, use the `o` option or the `OUTPUT` directive to specify an output filename.

Incremental Linking

You can use the output of the linker as input to subsequent linker sessions. To generate relocatable linked files for such incremental linking, invoke the linker with the `-r` option.

Linking a non-relocatable input file generates a warning message. For successful linking, non-relocatable input files must:

- have no unresolved external references
- be located at the same address as in previous linker invocations

Object Module Format Compatibilities

You can link b.out-format, COFF or ELF object files and libraries, in any combination. To determine a file format, the linker examines the first two bytes of the file. An unrecognized value indicates a linker-directive file.

This feature is useful when using third-party archives with the Intel-supplied runtime libraries and your application code. The runtime libraries are shipped in ELF *only* (effective with CTOOLS R5.0 and later). Each archive can have a different Object Module Format (OMF), and the linkage still completes without error.

When the linker generates a different output format than the input, the linker does not copy the debug information from the input file to the output file. For example, if you include a b.out OMF file in a linkage where the output file OMF is COFF, the linker does not copy any of the debug information from the b.out file to the output COFF file. Ideally, you should use one OMF consistently.

However, the following symbol information is translated and incorporated into the output OMF file:

- Leaf procedures
- System Procedures (sysprocs)
- Common (global uninitialized variables)
- Locals (statics) in any section (typically `.text`, `.data`, or `.bss`)
- Globals in any section
- Absolutes

The linker ignores any other type of symbol when the output OMF is different from the input, including:

- Type Information (e.g., x is a structure containing elements y and z).
- Line Information (e.g., function x , comes from file `x.c`, and ranges from lines 123-456. Line 235 is at ip 0x12345678).
- Weak Symbols. ELF supports a third granularity of symbol type. None of the other OMFs supports it.

Some OMFs are not as rich as others and therefore cannot accommodate the features of other OMFs. Some examples follow:

1. **Sections.** The b.out OMF accommodates *only* .text, .data, and .bss sections. If you create a b.out output file and attempt to use the linker to create a section with any other name, the linker terminates with an error. There is a workaround for this in the normal linker directive language. Suppose you have an ELF file called `elf_file.o` with a section called `Other_code_section`. You can use the following linker directives to include the contents of this section in the .text section of the b.out output file:

```
SECTIONS {
    .text : {
        elf_file.o(Other_code_section)
        ...
    }
    ...
}
```

Also, COFF accommodates only sections with names that are up to eight characters long, whereas ELF can have arbitrary length section names. When incorporating an ELF section into COFF output, the linker shortens the name to eight characters.

2. **Relocation types.** b.out does not support ELF's `R_960_SUB` relocation type. If you try to create a relinkable output file with one of these relocations in it, and the output OMF is b.out or COFF, an error occurs. Note, however, that *not* making a relinkable file does not create a problem. The linker relocates it, and throws the relocation directives away.
3. **System procedure (sysproc) indices.** The b.out OMF accommodates only system procedure indices that are greater than zero and less than 254. If you have a `.sysproc` symbol in a COFF or ELF symbol table whose system procedure index value exceeds the bounds that b.out supports, and you attempt to include this file in a b.out linkage, an error occurs.

Note that when a `.leafproc` crosses from ELF or COFF into `b.out`, the leaf entry point is produced from the call symbol. For example:

```
.leafproc x,L0
x:
    lda 0,g0
L0:
    ret
$ gas960e t.s
$ gld960 -r t.o -Fbout -o t.bout.o
$ gnm960 t.bout.o

Symbols from t.bout.o:

0x00000000 t x
0x00000004 t x$LF
```

Symbol `x` appears, and also symbol `x$LF` (which corresponds to the leaf entry point of `x`).

Link-time Optimization

When the linker performs relocation, it changes `callj/calljx` pseudo instructions into `bal/balx` instructions when the target of the `callj/calljx` is a leaf procedure. It also changes the instruction to a `calls` when the target is a system procedure index. In your assembly source, use the `callj` and `calljx` pseudo-instructions, to be replaced as shown in Table 7-2:

- When the pseudo-instruction argument is a leaf procedure, the linker substitutes a branch-and-link, which is faster than a call.
- For system procedures, the linker substitutes system calls, providing convenient access to a set of kernel services.
- To prevent these optimizations, use the `-o` option.

Table 7-2 Branch-and-link and System-call Optimization

Pseudo-instruction	Storage Class	Replaced By
callj	external, static	call
	leaf external	bal
	system call	calls
calljx	external, static	callx
	leaf external	balx
	system call	lda index, g13; calls g13

For a system call, the processor refers to a system-procedure-table index, provided as the `calls` argument:

- For an index from 0 to 31 with `callj`, the linker generates a `calls` with a literal index constant.
- For an index from 0 to 259 with `calljx`, the linker generates instructions to load the index into register `g13` and perform the call as follows:

```
lda index, g13
calls g13
```

Since register `g13` is used, avoid returning a structure longer than 16 bytes from a system procedure. Instead, return pointers to a structure.

The following cause out-of-range errors:

- using `callj` with an index greater than 31
- using `calljx` with an index greater than 259 for COFF programs
- using `calljx` with an index greater than 257, or less than 1, for b.out format programs

For link-time optimization, design any hand-coded assembly language leaf procedures for both call (`callj/calljx`) and branch-and-link (`bal/balx`) access. Then, the linker can optimize the call while protecting any indirect procedure accesses that are not recognized as optimizable.

The following example provides both branch-and-link and call access:

```
                .leafproc      _name, _second
_name:          lda            retlbl, g14
_second:       mov            g14, g13
# The subroutine appears here.
                bx              (g13)
retlbl:        ret
```

- Branch-and-link instructions place the address of the next instruction in g14 before branching.
- The lda instruction at the _name: label places the retlbl address of the ret instruction in g14.
- The first .leafproc argument, _name, is used as the call and callx entry point.
- The second .leafproc argument, _second, used as the bal and balx entry point.
- The branch destination is g13, whose contents are determined by the contents of g14, which vary depending upon the entry point. In either case, the routine returns correctly when the bx instruction is executed:
 - The lda instruction at _name places retlbl in g14 when the routine is entered by _name and by a call or callx instruction.
 - The g14 register contains the address of the instruction after the bal or balx when the routine is entered by _second and by a bal or balx instruction.

For more information on the callj and calljx pseudo-instructions, see the *i960® Processor Assembler User's Guide*.

Using calljx with the i960® Rx Processor

When using a `calljx` pseudo instruction with the `-ARP` or `-ARD` option, `calljx` uses a different syntax. For example, inserting a `calljx` instruction while using the `-AJD` setting might produce the following linker output depending upon whether the target is a default call, leaf procedure, or system call:

Default Call	Leaf Procedure	System Call
<code>callx _target</code>	<code>balx _target,g14</code>	<code>lda _sysprocIndex,g13</code> <code>calls (g13)</code>

When using a `calljx` with the new `-ARP` or `-ARD` option, `calljx` uses the syntax:

```
calljx _target, tmpreg
```

where `tmpreg` is a local or global register. This change results in the following sequences in the linker:

Default Call	Leaf Procedure	System Call
<code>lda _target,tmpreg</code> <code>callx (tmpreg)</code>	<code>lda _target,tmpreg</code> <code>balx (tmpreg),g14</code>	<code>lda _sysprocIndex,g13</code> <code>calls (g13)</code>

Notice that with the 80960Rx `calljx` format all three call types result in a three-word instruction sequence, whereas with other architectures the previous `calljx` format requires only two words.

Binding Profile Counters to Non-standard Sections

When compiling for two-pass compilation, the compiler places profile counters in your code. These are COMMON variables, but are allocated jointly; therefore, all profile counters occupy a contiguous stream of memory. By default, these profile counters are allocated to the same memory as other common variables, but they can be allocated to any memory using the wildcard linker section directive.

Environment Variables

Environment variables set default operating parameters, such as search paths and the target architecture. Define the environment variables before invoking the linker. UNIX users must define the environment variables with the `set` or `setenv` command at the operating system prompt, in script files, or in a bootup file such as `.cshrc`, `.login`, or `.profile`. Windows users define the environment variables at the operating system prompt, in batch files, in the `autoexec.bat` file, or using the Control Panel.

For more information on the environment variables used by the i960® processor assembler and software utilities, see the *i960® Processor Assembler User's Guide*. For more information on defining environment variables, see your host system documentation.

The linker requires you to define the following environment variables:

G960BASE	specifies base directory for invoking the linker as <code>gld960</code> .
I960BASE	specifies base directory for invoking the linker as <code>lnk960</code> .
G960ARCH	specifies the target-architecture libraries for the <code>gld960</code> invocation.
I960ARCH	specifies the target-architecture libraries for the <code>lnk960</code> invocation.
G960LIB	specifies an additional directory for the <code>gld960</code> -invocation library and directive-file search path.
G960LLIB	library and directive-file search path.
I960LIB	specifies additional directories for the <code>lnk960</code> -invocation.
I960LLIB	library and directive-file search path.

For more information on how setting these environment variables affects the linker, see the *Library Naming Conventions and Search Paths* section on page 7-21.

Library Naming Conventions and Search Order

The `labbr` option specifies an abbreviation for one of the six standard library types (standard ANSI, math, floating-point, C++ iostream library functions, 64 bit integer support). The linker combines the abbreviation for the type with the architecture option and big-endian and position-independent code and data options, if any, to generate a list of candidate library names.

```
abbrqual.a
libabbrqual.a
abbrarchqual.a
libabbrarchqual.a
```

`abbr` is the argument of the `l` option, one of:

<code>c</code>	which contains the standard ANSI C functions.
<code>m</code>	which contains the standard ANSI math functions.
<code>h</code>	which contains the accelerated floating-point functions for processors without on-chip floating-point support.
<code>i</code>	which contains C++ iostream library
<code>ll</code>	which contains a MON960 low-level library.
<code>u</code>	which contains 64 bit integer support functions

`qual` is null unless big-endian, PIC or PID options are specified on the linker command line. Then, `qual` is one of:

<code>_b</code> or <code>b</code>	which selects a big-endian library for Cx, Jx, and Hx applications.
<code>_p</code> or <code>p</code>	which indicates that the library contains position-independent data (PID).
<code>_e</code> or <code>e</code>	which indicates a PID and big-endian library for Cx, Hx, and Jx applications.

arch is the architecture option specified by the linker command line. It is one of:

<code>ca</code>	for i960® Cx, Hx, and Jx processor-based applications.
<code>jx</code>	for i960 Jx processor-tuned floating-point libraries.
<code>ka</code>	for i960 KA and SA processor-based applications.
<code>kb</code>	for i960 KB and SB processor-based applications.
<code>rp</code>	for i960 RP/RD processor-based applications.

The linker searches for each library name along a library search path determined by the linker invocation.

When you invoke the linker as `gld960`, the search path is:

1. any path given with the `L` option
2. any path given with the `SEARCH_DIR` directive
3. the path given with the `G960LIB` environment variable
4. the path given with the `G960LLIB` environment variable
5. the path given with the `G960BASE` environment variable
6. the current working directory

When you invoke the linker as `lnk960`, the search path is:

1. any path given with the `L` option
2. any path given with the `SEARCH_DIR` directive
3. the path, if any, given with the `I960LIB` environment variable
4. the path, if any, given with the `I960LLIB` environment variable
5. the `lib` subdirectory of the path given with the `I960BASE` environment variable
6. the current working directory

The following example shows a linker search path, using the slash (/) UNIX directory syntax:

```
lnk960 -L/ffs/qqq -lxyz objects.o -Tpath2
```

The `path2.ld` directive file contains `SEARCH_DIR(/abc)`.

For the default (KB) architecture, the linker constructs the following library filenames:

```
xyz.a
libxyz.a
xyzkb.a
libxyzkb.a
```

The linker searches the following directories:

```
/ffs/qqq/xyz.a
/abc/xyz.a
$I960LIB/xyz.a
$I960LLIB/xyz.a
$I960BASE/lib/xyz.a
./xyz.a
/ffs/qqq/libxyz.a
/abc/libxyz.a
$I960LIB/libxyz.a
$I960LLIB/libxyz.a
$I960BASE/lib/libxyz.a
./libxyz.a
/ffs/qqq/xyzkb.a
/abc/xyzkb.a
$I960LIB/xyzkb.a
$I960LLIB/xyzkb.a
$I960BASE/lib/xyzkb.a
./xyzkb.a
/ffs/qqq/libxyzkb.a
/abc/libxyzkb.a
$I960LIB/libxyzkb.a
$I960LLIB/libxyzkb.a
$I960BASE/lib/libxyzkb.a
./libxyzkb.a
```

Library Search Order When i960® Rx Architecture Is Selected

When a non-i960® Rx architecture is specified, the linker searches first for architecture-neutral libraries, then for architecture-specific libraries. For example, when the linker looks for the i960 KA processor `libc` library, it first tries to find `libc.a` and, if the library is not found, the linker looks for `libcka.a`. Because files targeted for the i960 Rx processor require target-specific libraries, the linker looks first for architecture-specific libraries (e.g., `libcrp.a`), and, if those libraries are not found, the linker looks for architecture-neutral libraries (e.g., `libc.a`).

Linker Options Reference

This section describes the linker command-line options in alphabetical order.

A: Architecture

*Selects libraries;
identifies instruction set*

Aarchitecture

architecture is SA, SB, KA, KB, CA, CA_DMA, CF, JA, JD, JF, JT, HA, HD, HT, RD, RP, RM, RN, VH, CORE0, CORE1, CORE2, or CORE3

Discussion

Specifying the architecture:

- selects the libraries for your target i960 processor
- identifies the instruction set used in the input object files

To specify the architecture, use the **A** option. This overrides any `OUTPUT_ARCH` directive and the `I960ARCH` or `G960ARCH` environment variable.

Some pairs of **A** arguments have identical effects:

- **SA** is the same as **KA**
- **SB** is the same as **KB**
- **CA** is the same as **CF** and **CA_DMA**

You can prepare several levels of default values for the architecture and standard libraries. Omitting **A** uses the architecture specified by `OUTPUT_ARCH` in the linker-directive file. Omitting both **A** and `OUTPUT_ARCH` uses the default architecture for the linker invocation and the architecture environment variables (described in the *Library Naming Conventions and Search Paths* section on page 7-21):

- Invoking the linker with `lnk960` uses the `I960ARCH` environment variable.
- Invoking the linker with `gld960` uses the `G960ARCH` environment variable.

- With I960ARCH or G960ARCH undefined, the default architecture is KB, regardless of the invocation command.

Specifying A with no valid argument causes a fatal error.

New Architecture Options

The linker now accepts -AJT, -ARP, -ARD, -ARM, -ARN, and -AVH architecture switches or environment variable settings. The following table shows the input/output compatibilities of all supported architectures.

Table 7-3 Supported Input/Output Architecture Combinations

		Output									
		SA/ KA	SB/ KB	Cx	VH/ Jx	Hx	Rx	CORE0	CORE1	CORE2	CORE3
I n p u t	SA/KA	C	C	NA	NA	NA	NA	NA	NA	NA	NA
	SB/KB	NA	C	NA	NA	NA	NA	NA	NA	NA	NA
	Cx	NA	NA	C	NA	NA	NA	NA	NA	NA	NA
	Jx	NA	NA	NA	C	C	NA	NA	NA	C	NA
	Hx	NA	NA	NA	NA	C	NA	NA	NA	NA	NA
	Rx	NA	NA	NA	C	C	C	C	NA	C	NA
	CORE0	NA	NA	NA	C	C	C	C	NA	C	NA
	CORE1	C	C	C	C	C	NA	NA	C	C	C
	CORE2	NA	NA	NA	C	C	NA	NA	NA	C	NA
	CORE3	NA	NA	C	C	C	NA	NA	NA	C	C

C = compatible.

NA = incompatible. Warning issued.

Libraries

The linker uses the architecture-specific standard libraries for an `HLL` directive without arguments. The provided directive files, such as `eva.ld`, contain the appropriate library directives. The designated architecture also affects the library names generated from the `l` (library) option.

Example

The following links `proto.o` for a KA target.

```
lnk960 -AKA proto.o
```

Related Topic

`l` (Library Input)

B, T: Section Start Address

Assigns a section starting address

`Bsection addr`

$$\begin{matrix} \text{T} \left\{ \begin{array}{l} \text{bss} \\ \text{data} \\ \text{text} \end{array} \right\} \text{addr} \end{matrix}$$

`section` is a section name. The space between `section` and `addr` is required.

`bss` identifies the `.bss` section.

`data` identifies the `.data` section.

`text` identifies the `.text` section.

`addr` is a hexadecimal integer constant for `T`, or an octal, decimal, or hexadecimal expression for `B`.

Put no space between `T` and `bss`, `data`, or `text`. Do not start the `bss`, `data`, or `text` argument to `T` with a dot (`.`).

A space between `B` and the section name is optional. Use the full section name, including any leading dot.

Discussion

To specify a section starting address, use `B`, overriding any other default or directive-file section starting address. For `addr`, you can use:

- an octal constant starting with `0`.
- a decimal constant starting with any digit other than `0`.
- a hexadecimal constant starting with `0x`.
- an expression that can contain octal, decimal, and hexadecimal constants and symbols defined in linker directive files or with the `defsym` option.

For backward compatibility, `T` is supported, with the following restrictions:

- Specify `addr` as a hexadecimal constant. Expressions are not evaluated. Regardless of the leading character or digit, `T` interprets the address as hexadecimal.
- Use `T` for only the `.text`, `.data`, and `.bss` sections.

Examples

- The following starts `.data` at the address `1000` hexadecimal and starts the section named `mydata` at the address `24` hexadecimal:

```
lnk960 -Tdata 0x1000 -Bmydata 0x24 file1.o file2.o
```

- When `B` and `T` locate the same section name, `B` overrides `T`. The following starts `.text` at `0xc`:

```
lnk960 -Ttext 0xf0 file1.o file2.o -B.text 0xc
```

- The starting address of `sect1` in the following command can be expressed as `012750` octal, `5608` decimal, or `0x15e8` hexadecimal:

```
lnk960 -Bsect1 0x1000+1000+01000 file1.o file2.o
```

C: Startup Alternative

*Suppresses any
STARTUP directive in
the linker directive file*

C

Discussion

By default, the first object file or library specified on the command line is linked first. To link a different file first, use `C`. The `C` option overrides the `STARTUP` directive and returns to the default.

Example

For example, the `my-targ.ld` file contains a `STARTUP(crtest.o)` directive. The following links `newstart.o` first instead of `crtest.o`:

```
lnk960 -C -Tmy-targ newstart.o file1.o
```

c: Circular Library Search

*Searches libraries
circularly*

c

Discussion

By default, the linker processes libraries in order, reading from left to right on the command line. In most cases, this approach works well.

Occasionally, however, libraries contain circular references. In such cases, you can use the linker's `c` option to search libraries iteratively to resolve these references. This does, however, slightly change the semantics of

links; formerly undefined symbols may be defined by other libraries in the loop.

Example

`lib1` and `lib2` interrelate as follows:

lib1 defines	references	object file
x	a	x.o
y		y.o
lib2 defines	references	object file
a	y	a.o

If you link using:

```
gld960 -ux lib1.a lib2.a
```

the linker opens `lib1.a` first, finds symbol `x` resolved in the `x.o` object file, and reads it in. It finds no other references to symbols it needs, so it closes `lib1.a`.

The linker then opens `lib2.a` and resolves the symbol `a` referenced in the `x.o` file (from `lib1.a`). It reads in the `a.o` object file. It finds no other symbols it can resolve, so it closes `lib2.a`.

However, symbol `y` is left unresolved, because it is defined in `lib1.a` and referenced in `a.o` (from `lib2.a`). This is a circular definition.

The `c` option resolves circular definitions by iterating through the list of libraries and fetching object files from them as needed for each loop. The search is complete when:

- there are no more undefined symbols, OR
- neither the number of global symbols nor the number of undefined symbols has changed from the previous iteration.

D: Inhibit CAVE section compression

Prevents the linker from compressing CAVE sections

D

Discussion

By default the linker compresses any sections set up for Compression Assisted Virtual Execution (CAVE) by the user. Using the `D` option prevents the linker from performing this processing. For more information on CAVE, see the *i960® Processor Compiler User's Guide*.

d: Define Common Symbol Space

Allocates common symbols to .bss even when doing -r links

`dc`
`dp`

`c` and `p` have identical effects.

Discussion

To assign common-symbol space in an output file linked with the `-r` (Relocation) option, use `dc` or `dp`. This option has the same effect as the `FORCE_COMMON_ALLOCATION` directive. (It places common symbols by default into the `.bss` section.)

The final link automatically allocates space for common symbols.

Example

The following assigns space in the `.bss` section of `a.out` for common symbols and retains relocation information for later re-linking:

```
lnk960 -rdc dcomm.o file1.o
```

defsym: Define a Symbol

Defines an absolute symbol

```
defsym name=expr
```

name names the symbol.

expr initializes the symbol.

Discussion

To define a symbol on the linker command line, use `defsym`. You can reference the symbol in your source text, in a directive file, or on the linker command line.

Example

The following resolves any references to `filenum1` in `file1.o` or `file2.o`. Its value is zero (absolute).

```
lnk960 -defsym filenum1=0 file1.o file2.o
```

Related Topic

R (Read symbols only)

e: Entry Point

*Defines the primary
entry point*

*e*symbol

symbol

is a symbol name in a text-type section in the output file.

Discussion

To define the primary entry-point symbol in the output file, use *e*.

The linker uses the following order of precedence to select an entry point:

1. *e* on the command line
2. with *e* unspecified, `ENTRY` in the linker directive file
3. with *e* and `ENTRY` unspecified, the first appearance of `start` or `_main` in your program
4. with `start` and `_main` undefined, the first address in `.text`

Example

The following command links `file.o` for execution on a Cx target and specifies the symbol `midpoint` as the entry point:

```
lnk960 -Tmepcx -e midpoint file.o
```

F: Format

*Specifies the COFF,
ELF or b.out format for
the output*

<code>Fcoff</code>	specifies COFF output.
<code>Felf</code>	specifies ELF output.
<code>Fbout</code>	specifies b.out output.

Discussion

To specify an output format, use the `F` option.

`gld960` or `gld960 -Fbout` generates b.out format output. The `Fbout` option is not valid when the linker is invoked with `lnk960`.

`gld960 -Fcoff` or `lnk960` generates COFF output.

`gld960 -Felf` or `lnk960 -Felf` generates ELF output.

The output format generates the default output filename (see the *Linker Invocation* section and the `o` option in this section). The default format for `gld960` is b.out; the default format for `lnk960` is COFF.

Example

The following generates a COFF program:

```
gld960 -Fcoff file1.o file2.o
```

f: Fill

Sets the fill value for unused memory in an output section

fvalue

value

is a two-byte hexadecimal constant in C-style notation.

Discussion

Use this options to initialize blocks of memory in sections of an output file.

You can prepare several levels of default fill values:

- A `FILL` directive inside a section (valid only for parts of the section defined after `FILL` is encountered) definition is used first.
- With no fill value inside a section definition, the linker uses the `FILL` directive at the end of the current section definition, inside the `SECTIONS` directive.
- With no fill value defined in the directive file, the linker uses the `f` option.
- With both `f` and `FILL` omitted, the linker uses a 0 fill value.

Filler is used to ensure alignments between input sections:

```
a.s          b.s          .text 0
.align 4     .align 4     lda 0,g0
lda 0,g0     lda 0,g0     Filler
lnk960 a.o b.o          lda 0,g0
```

Example

The following command links `file1.o`, `file2.o`, and `file3.o` to produce an executable image named `a.out`. The linker places `0xFFFF` in all gaps between the input sections in the output file.

```
lnk960 -f 0xFFFF file1.o file2.o file3.o file.o xxx.ld
SECTIONS {
    one: {
        first.o (.text)
        . += 0x1000; /*filler*/
        second.o (.text)
    }
}
```

G: Big-endian Target

*Produces a COFF or
ELF file for a big-
endian target*

G

Discussion

To link big-endian instructions and data, invoke the linker and:

- Specify `G`, to select the big-endian search path and libraries, as described in the *Library Naming Conventions and Search Paths* section on page 7-21.
- Specify `Fcoff` or `Felf`, for COFF or ELF output.
- Select the i960 Cx, Jx, or Hx architecture.



NOTE. *Big-endian code is supported for COFF or ELF on the i960 Cx, Jx and Hx processors only. Memory regions must be either all big-endian or all little-endian. The linker emits warnings when you attempt to mix big- and little-endian code.*

Example

The following links with a user-defined library (abbreviated as `myca`) for a big-endian CA target. The linker uses the `_b` or `b` qualifiers first when searching for the library indicated by `-lmyca`.

```
gld960 -ACA -G -Fcoff fcab.o fcag.o -obigca.o -lcg -lm -lmyca
```

The objects must have been produced using the assembler's `G` option.

gcdm: Decision maker

*Invoke `gcdm960`
optimization decision
maker*

`gcdm`

Discussion

See Chapter 6, *gcdm Decision Maker Option*, in your compiler manual for more information on this option.

h: Help

Displays help information

h

Discussion

To display help information for the linker, use the `-h` option.

H: Sort common symbols

Sorts common symbols based on size.

H

Discussion

To use the linker to sort common symbols based on size, use the `-H` option. For each input file in the linkage, the common symbols defined in that file are sorted based on the size of the symbol.

J: Compress

*Merges duplicated tags
from COFF symbol
tables, compresses
string table*

J

Discussion

This option merges duplicated COFF symbol tags from output symbol tables.

Linking with the J option eliminates such duplicated tags. The resulting symbol table has tag indices that cross .file scope boundaries. The file has `F_COMP_SYMTAB` ORed into the flags of the file header structure (see `coff.h`). The string table is also compressed with this option.

L: Library Search Path

*Changes the path for
library searches*

Ldir

dir is a directory name.

Discussion

To extend the linker search path (described in the *Library Naming Conventions and Search Paths* section on page 7-21), use L. You can use L multiple times on the command line. The L option has the same effect as `SEARCH_DIR`, but directories specified with L are searched before directories specified with the `SEARCH_DIR` directive.

I: Library Input

Specifies an input library

`labbr`

`abbr`

is an abbreviated form of a library name. Only one `abbr` can accompany each `l`.

Discussion

See the *Library Naming Conventions and Search Paths* section on page 7-21 for information on this option.

Only the first filename found is used. Once closed, a library is reopened only when specified again on the command line or in a linker directive file, or if the `-c` is used.

You can use the `u` option to create an unresolved reference to a symbol in the desired library member before specifying the library.



NOTE. *Because the linker processes libraries and files in order, the appearance order of the `l` option on the command line is significant. For example, `gld960 f.o -lh` differs from `gld960 -lh f.o`, and `gld960 ... -lh -lc` differs from `gld960 ... -lc -lh`.*

Examples

In the following example:

- Input `file1.o` refers to the `ABC` symbol, defined in member 0 of `libckb.a`.
- Input `file2.o` refers to the `XYZ` symbol, defined in member 0 of `liba.a`.
- Both input files refer to the `FCN` external function, defined in member 1 of both libraries.

The command line is:

```
lnk960 file1.o -la file2.o -lc
```

The `FCN` references are satisfied by `liba.a`, member 1; `ABC` is obtained from `libckb.a`, member 0; and `XYZ` remains undefined, since the library `liba.a` is searched before `file2.o` is specified.

To repair this, consider changing the command line to:

```
lnk960 file1.o -u_XYZ -la file2.o -lc
```

You can create an unresolved reference from the command line with the `u` option. Such references link archive members needed to resolve the undefined symbol, even when the input does not explicitly reference the symbol. The following command creates an undefined symbol, called `rou1`, in the global symbol table:

```
lnk960 -u rou1 file1.o -la
```

The linker extracts the first member of library `liba.a` that defines `rou1`. With no other references to `liba.a` members, the linker would link only the member resolving `rou1`.

M: Multiple Definition Warning

Included for backwards compatibility, No effect

M

Discussion

This option is supported for backwards compatibility, but has no effect.

Related Topic

t (Multiple definition warning)

m: Memory Map

Provides a memory map of the linked executable

m

Discussion

To write a memory map of the linked executable to `stdout`, specify `m`, which lists:

- the symbol locations
- the global common storage allocation

For information on `stdout`, see your host system documentation. You can redirect map information to a file using the `N` option.

Related Topic

`N` (Name memory map file)

N: Name Memory Map File

Specifies a filename for writing the memory map

Nfile

Discussion

Redirects the linker memory map to the specified file *file*. When this option is not specified but the `m` option is given, map information is written to standard output. This option allows you to separate the linker map from other information written to standard output, such as verbose messages and warnings.

Example

The following command links `t.o` with verbose messages, and redirects the linker map to a file `mapfile`. Verbose messages are still sent to standard output.

```
lnk960 -m -Nmapfile t.o -v
```

Related Topic

`m` (Memory map)

n: Noinhibit Output

*Writes an output file
regardless of errors*

n

Discussion

To produce an output file even when the linker encounters non-fatal errors, specify n. By default, any error suppresses the output file.

Example

The following command produces an output file named `haserr.o` regardless of non-fatal errors:

```
lnk960 -n proto.o -ohaserr.o
```

O: Optimization of Calls Inhibited

*Inhibits branch-and-link
or system call
optimizations*

O { b | s }

b

inhibits the replacement of `callj` and `calljx` pseudo-operations with branch-and-link instructions.

s

inhibits the replacement of `callj` and `calljx` pseudo-operations with system call instructions. call instructions remain.

Discussion

To inhibit branch-and-link optimizations, specify `ob`. To inhibit system call optimizations, specify `os`. By default, the linker performs both types of call optimization.

Note that if a function declared with `#pragma systemproc` lacks a function definition, `-os` causes the linker to issue a fatal error.

Example

The following command inhibits system call optimizations but allows branch-and-link optimizations:

```
lnk960 -os proto.o
```

o: Output Filename

Names the output object file

ofilename

filename names the output file. You can include a full or partial pathname.

Discussion

To specify an output filename other than the default, use `o`. The default output filenames are:

<code>a.out</code>	for COFF output
<code>b.out</code>	for b.out format output
<code>e.out</code>	for ELF output

Example

The following command links `file.o`, creating `proto.out` in the `/testdir` directory:

```
lnk960 -Texv -o /testdir/proto.out file.o
```

Related Topic

F (Format)

P: Profiling

Puts profiling code in the linker output to support the two-pass optimizing compiler

P

Discussion

This option adds the profiling startup code used by the compiler. This option is useful only when combined with `-r`. By default `P` is not included when using the `r` option

Note that the linker does not properly handle a file with `cc_info` (two-pass profiling information generated by the compiler) without the presence of a `.text` section.

Example

The following command links for profiling optimization and makes the output relocatable:

```
gld960 -P -r file.o
```

p: Position-independence

Marks the linker output file as position-independent

p { b | c | d }

Discussion

To link and mark the output file for position-independent code or data, invoke the linker with `lnk960` and specify `p`, as follows:

<code>pb</code>	selects libraries with position-independent code and data.
<code>pc</code>	selects libraries with position-independent code. Currently, all the libraries provided with your i960® processor software toolset contain position-independent code.
<code>pd</code>	selects libraries with position-independent data.

For more information on library selection, see the *Library Naming Conventions and Search Paths* section on page 7-21 and the `L` option in this section.

By default, files are linked as position-dependent. If you use these switches and the files are not marked as position independent, the linker issues a warning message.

For information on generating position-independent code and data, see your compiler user's guide. For information on marking unlinked object files for position independence, see the assembler user's guide. For information on the position-independent and position-dependent libraries, see the library supplement.

When the linker directive file contains `HLL()`, the linker chooses default libraries according to the position-independent flag.

R: Read Symbols Only

Includes only the symbols from an object file

R

Discussion

To read all the symbol names and addresses from an input object file, specify `R`. The rest of the input file is not relocated or included in your linked output. With `R`, your output file can refer symbolically to non-relocatable locations defined in other programs.

Example

The following command links only symbols from `file1.o` and includes all of `file2.o` in the linked output:

```
lnk960 -R file1.o file2.o
```

r: Relocation

*Keeps relocation entries
in the output object file*

r

Discussion

With the `r` option, relocation entries remain in the output object file for a subsequent linker call, and the linker issues no warnings about unresolved references.

Relocation requires symbol table entries that you can remove with the `s` option. The linker accepts no command line containing both the `-r` and `-s` options.

Example

In the following incremental links, the first invocation links `file1.o` and `file2.o` to produce the relocatable output file `f1.out`. The second links `file3.o` and `file4.o` to produce `f2.out`. The third links the two relocatable files to produce `done.out` and writes a link map to `stdout`.

```
lnk960 -r -o f1.out file1.o file2.o
lnk960 -r -o f2.out file3.o file4.o
lnk960 -m -o done.out f1.out f2.out
```

Related Topics

`x`, `X` (Compress)
`s` (Strip)

`dc`, `dp` (Define common-symbol space)

S, s: Strip

*Removes debugging or
symbolic information
from the object file*

`S`

`s`

Discussion

For a smaller output file, strip symbol information with `s` (lower case), removing:

- the line number entries
- the symbols
- the symbol-table information
- In ELF output, this eliminates all non-allocated sections (e.g., `.debug_info`, `.debug_line`).

Using `S` (uppercase) retains the symbol table but removes debug symbols. This option is supported in COFF and ELF only. In ELF, this removes all non-allocated sections (`.debug*` sections). By default, all information remains in the output file.

Since relocation requires the symbol table, using `s` with the relocation option (`r`) terminates the linker with an error.

Related Topics

`x`, `X` (Compress)
`r` (Relocate)

T: Target

Searches for the linker directive file in the linker search path

`Tfilename`

`filename` is the linker directive filename. You need not specify a `.ld` extension.

Discussion

To find a linker directive file in a directory other than the current one, specify the file with `T`. Providing the directive filename without `T` limits the linker to searching in the current directory.

The linker searches for both `filename` and `filename.ld`.

With `T`, the linker searches for the directive file along the search path described in the *Library Naming Conventions and Search Paths* section on page 7-21.

The `T` option has the same effect as a `TARGET` directive.

For information on the linker command language used in linker directive files, see Appendix A.



NOTE. *You must add the .ld extension when specifying any directive filenames that are the same as the standard section names:*

text.ld use Ttext.ld

data.ld use Tdata.ld

bss.ld use Tbss.ld

You cannot use the T option to find a directive file named text, data, or bss. You can use the names text.ld, data.ld, or bss.ld, but you must append the filename and its extension when you use the T option.

Related Topics

A (architecture)

B

L (library directory)

Ttext, Tdata

t: Suppress Multiple Definition Symbol Warnings

Suppresses warning of multiple symbol definitions.

t

Discussion

Use this option to suppress warnings of multiple symbol definitions, even if they differ in size

Related Topic

w (Warnings)

u: Unresolved Symbol

*Places unresolved
external symbol in the
symbol table*

usymbol

symbol names the symbol.

Discussion

When creating programs of libraries only, such as run-time libraries, build the symbol table with `u`. This option introduces an unresolved external symbol into the output file symbol table. The linker resolves the reference with the first module defining the symbol. This option is useful when libraries are to be traversed in an order that prevents your application from linking.

Example

The following example shows how to fetch the closure of components of `printf`.

```
lnk960 -rvu printf -oprintf.o -lc
```

v: Verbose

Displays linker progress

v

Discussion

To display the files sought by the linker as the linker search progresses, specify v. The search messages appear on `stdout`.

V, v960: Version

*Displays the linker
version number and
creation date*

V

v960

Discussion

To display a sign-on message on `stdout` during linking, use v. After displaying the message, the linker continues processing.

To display the message without linking, use v960. You need not provide any other input. After displaying the message, the linker stops.

The message includes the version number of the linker, and the date and time it was created.

W: Warnings

Suppresses warnings

w

Discussion

The linker provides warning messages about non-standard conditions arising during the link. Using w suppresses the warnings.

Related Topic

T (Suppress Multiple Definition Symbol Warnings)

X, x: Compress

*Omits local symbols
from the output symbol
table*

x

x

Discussion

Delete local symbols from the output symbol table as follows:

x removes all local symbols.

x removes all local symbols beginning with L or a dot (.).

When generating a compressed output file, you can also remove symbolic information with the strip option (s).

By default, all information remains in the output symbol table for symbolic debugging.

Related Topic

S, s (Strip)

y: Trace Symbol

*Traces the specified
symbol*

*y**symbol*

symbol

identifies the symbol.

Discussion

The linker traces the symbol *symbol*, indicating each file where it appears, its type, and whether the file defines or references it. You can trace multiple symbols by using multiple *y* options. If *symbol* comes from a C program, you must precede it with an underscore.

Z: Program database

*Specifies location of
program database*

z PDB_directory

Discussion

Use this option to specify the location of the program database. For information on the program database, used when performing whole-program or profile-driven optimization, see your compiler manual.

z: Time Stamp Suppression

*Suppresses the time
stamp in the COFF
output file*

z

Discussion

For COFF files, the linker notes the current time and date in the output-file header. To put Time Zero in place of the current time stamp, specify *z*. Time Zero is 4:00, 31 December, 1969.



NOTE. *Neither b.out format nor ELF files have a time stamp.*

Macro Processor (*mpp960*)

The *mpp960* macro processor copies its input to its output, expanding macros as it goes. The macros are either built-in or user-defined and can take any number of arguments. *mpp960* has built-in functions for including named files, running UNIX commands, performing integer arithmetic, manipulating text in various ways, doing recursion, and performing other tasks. *mpp960* can be used as a stand-alone macro processor or as a front-end to a compiler or assembler.

mpp960 is compatible with the UNIX System V, Release 3 *m4* utility, with some minor differences. See the *Compatibility with Other Macro Processors* section in this chapter for more details.

mpp960 Message Prefixes

This chapter contains many examples of *mpp960* input and output. Output from *mpp960* is prefixed by the string `=>`. For example:

```
=>Output line from mpp960
```

Error messages are prefixed by the string `error-->`

```
error-->and an error message
```

mpp960's predefined macros are described by a prototype call of the macro using descriptive names as arguments.

```
regexp(string, regexp, [replacement])
```

All *mpp960* macro arguments are strings, but some strings are interpreted as numbers, filenames, or regular expressions.

The [] characters around the third argument shows that this argument is optional — when it is left out, it is taken to be the empty string. An ellipsis (...) last in the argument list means that any number of arguments can follow.

Invoking mpp960

The format of the `mpp960` command is:

```
mpp960 [-option]... [macro-definition]... [input-file]...
```

where *-option* is any of the following:

<i>Dname</i>	Enters name into the symbol table, before any input files are read. When =value is missing, the value is taken to be the empty string. The value can be any string, and the macro can be defined to take arguments just as if defined from within the input.
<i>dflags</i>	Sets the debug-level according to the <i>flags</i> . The debug-level controls the format and amount of information presented by the debugging functions. <ul style="list-style-type: none"> a Shows the actual arguments in each macro call. c Shows several trace lines for each macro call. e Shows the expansion of each macro call, if it is not void. f Shows the name of the current input file in each trace output line. i Prints a message each time the current input file is changed. l Shows the current input line number in each trace output line.

	p	Prints a message when a named file is found through the path search mechanism, giving the actual filename used.
	q	Quotes actual arguments and macro expansions in the display with the current quotes.
	t	Traces all macro calls made in this invocation of <i>mpp960</i> .
	x	Adds a unique macro call id to each line of the trace output.
	v	Shorthand for all the debug flags.
<i>efile</i>		Redirects debug and trace output to the named file. Error messages are still printed on the standard error output.
G		Suppresses all <i>mpp960</i> extensions that are not supported by the UNIX System V <i>m4</i> tool.
<i>Hn</i>		Sets the size of the internal hash table for symbol lookup <i>n</i> entries. The number should be prime. The default is 509 entries.
<i>Idir</i>		Makes <i>mpp960</i> search <i>dir</i> for included files that are not found in the current working directory.
i		Makes this invocation of <i>mpp960</i> interactive. This means that all output is unbuffered and interrupts are ignored.
<i>lnum</i>		Restricts the size of the output generated by macro tracing to <i>num</i> bytes.
<i>Nn</i>		Allows for up to <i>n</i> diversions to be used at the same time. The default is 10 diversions.
Q		Suppresses warnings about missing or superfluous arguments in macro calls.

s	Generates synchronization lines for use by the C preprocessor or other similar tools. This is useful, for example, when mpp960 is used as a front end to a compiler. Source filename and line number information is conveyed by lines of the form <code>#line <i>linenum</i> "<i>filename</i>"</code> that are inserted as needed into the middle of the input (but always on complete lines by themselves). Such lines mean that the following line originated or was expanded from the contents of input file <i>filename</i> at line <i>linenum</i> . <i>filename</i> may be omitted when the filename did not change from the previous synchronization line.
V	Displays the version number of the program.
v960	Displays the version number and exits.
B, S, T	Provides for compatibility with UNIX System V m4, but these options do nothing in this implementation.

Macro definitions and deletions can be made on the command line, by using the `D` and `U` options. They have the following format:

<code>Dname [=value]</code>	Enters <i>name</i> into the symbol table before any input files are read. When <i>=value</i> is missing, the value is taken to be the empty string. The <i>value</i> can be any string, and the macro can be defined to take arguments, just as if defined from within the input.
<code>Uname</code>	Deletes any predefined meaning <i>name</i> might have. Only predefined macros can be deleted in this way.
<code>tname</code>	Enters <i>name</i> into the symbol table as undefined but traced. The macro is consequently traced from the point it is defined.

The remaining arguments on the command line are taken to be input filenames. If no names are present, standard input is read. A filename of - is also taken to mean standard input.

The input files are read in the sequence given. The standard input can be read only once, so the filename - should appear only once on the command line.

Lexical and Syntactic Conventions

mpp960 separates its input into *tokens*. A token is either a name, a quoted string, or any single character that is not a part of either a name or a string. Input to *mpp960* can also contain comments.

Names

A name is any sequence of letters, digits, and the underscore character (`_`), where the first character is not a digit. If a name has a macro definition it is subject to macro expansion.

Examples of legal names are: `foo`, `_tmp`, and `name01`.

Quoted Strings

A quoted string is a sequence of characters surrounded by quotes; the number of start and end quotes within the string must balance. The so-called start and end quote characters are the backquote (```) and apostrophe (`'`), respectively. The value of a string token is the text, with one level of quotes stripped off. Thus, `` `` is the empty string and `` `quoted` `` is the string: ``quoted``.

The quote characters can be changed at any time using the built-in macro `changequote`.

Other Tokens

Any character that is neither a part of a name nor part of a quoted string is a token by itself.

Comments

Comments in mpp960 are normally delimited by the characters # and newline. All characters between the comment delimiters are ignored, but the entire comment (including the delimiters) is passed through to the output.

Comments cannot be nested, so the first newline after a # ends the comment. The begin comment character can be included in the input by quoting it.

The comment delimiters can be changed to any string at any time, using the built-in macro `changecom`.

How to Invoke Macros

This section describes macro invocation, macro arguments and how macro expansion is treated.

Macro Invocation

Macro invocations have one of these forms:

macroname

which is a macro invocation without any arguments, or:

macroname(arg1, arg2, ..., argN)

which is a macro invocation with *N* arguments. Macros can have any number of arguments. All arguments are strings, but different macros might interpret the arguments in different ways.

The opening parenthesis must follow the *macroname* directly, with no spaces in between. If it does not, the macro is called with no arguments at all. For a macro call to have no arguments, the parentheses must be left out. The macro call:

```
macroname ( )
```

is a macro call with one empty string argument, rather than a call with no arguments.

Macro Arguments

A name that has a macro definition is expanded as a macro. If the name is followed by an opening parenthesis, the arguments are collected before the macro is called. If too few arguments are supplied, the missing arguments are taken to be the empty string. If there are too many arguments, the excess arguments are ignored.

Normally, *mpp960* issues warnings when a built-in macro is called with an inappropriate number of arguments, but it can be suppressed with the `Q` command line option. For user defined macros, there is no check of the number of arguments given.

Macros are expanded normally during argument collection, and whatever commas, quotes and parentheses that might show up in the resulting expanded text defines the arguments as well. Thus, if `foo` expands to `,b,c`, the macro call:

```
bar(a foo,d)
```

is a macro call with four arguments: `a`, `b`, `c` and `d`.

Quoting Macro Arguments

Each argument has leading unquoted white space removed. Within each argument, all unquoted parentheses must match. For example, if `foo` is a macro:

```
foo(( ) (' ') (' '))
```

is a macro call, with one argument, whose value is `() (() (`.

It is common practice to quote all arguments to macros, unless you are sure you want the arguments expanded. To use this convention, you would change the above command to:

```
foo(`() () `')
```

Macro Expansion

When any arguments to a macro call have been collected, the macro is expanded and the expansion text is pushed back unquoted onto the input and reread. The expansion text from one macro call might therefore result in more macros being called, if the calls are included, completely or partially, in the first macro call's expansion.

Taking a very simple example, if `foo` expands to `bar`, and `bar` expands to `Hello world.`, the input:

```
foo
```

expands first to `'bar'`, and when this is reread and expanded, into:

```
Hello world.
```

How to Define New Macros

Macros can be defined, redefined, and deleted in several different ways. It is also possible to redefine a macro without losing a previous value. Previous values can be brought back at a later time.

Defining a Macro

The normal way to define or redefine macros is to use the built-in `define`:

```
define(name, expansion)
```

which defines *name* to expand to *expansion*.

The expansion of `define` is void.

The following example defines the macro `foo` to expand to the text "Hello World."

```
define('foo', 'Hello world.')
=>
foo
=>Hello world.
```

The empty line in the output is there because the newline is not a part of the macro definition and it is consequently copied to the output. You can avoid this by using the `dn1` macro.

Arguments to Macros

Macros can have arguments. The N th argument is denoted by $\$n$ in the expansion text, and is replaced by the N th actual argument, when the macro is expanded. Here is an example of a macro with two arguments. It simply exchanges the order of the two arguments.

```
define('exch', '$2', '$1')
=>
exch(arg1, arg2)
=>arg2, arg1
```

This can be used, for example, if you like the arguments to `define` to be reversed.

```
define('exch', '$2, $1')
=>
define(exch('`expansion text`', '`macro`'))
=>
macro
=>expansion text
```

For an explanation of the double quotes, see *Quoting Macro Arguments*. `mpp960` allows the number following the $\$$ to consist of one or more digits, allowing macros to have any number of arguments.

As a special case, argument zero, \$0, is always the name of the macro being expanded.

```
define('test', `Macro name: $0`)  
=>  
test  
=>Macro name: test
```

If you want quoted text to appear as part of the expansion text, remember that quotes can be nested in quoted strings. Thus, in:

```
define('foo', `This is macro 'foo'.`)  
=>  
foo  
=>This is macro 'foo'.
```

The `foo` in the expansion text is not expanded, since it is a quoted string, and not a name.

Special Arguments to Macros

There is a special notation for the number of actual arguments supplied and for all the actual arguments. The number of actual arguments in a macro call is denoted by \$# in the expansion text. Thus, a macro to display the number of arguments given can be:

```
define('nargs', ` $# `)  
=>  
nargs  
=>0  
nargs()  
=>1  
nargs(arg1, arg2, arg3)  
=>3
```

The notation \$* can be used in the expansion text to denote all the actual arguments, unquoted, with commas in between. For example:

```
define('echo', ` $* `)  
=>  
echo(arg1, arg2, arg3 , arg4)  
=>arg1,arg2,arg3 ,arg4
```

Use the notation `$@` when each argument should be quoted. It is just like `$*`, except that it quotes each argument. This is a simple example:

```
define('echo', '$@')
=>
echo(arg1, arg2, arg3 , arg4)
=>arg1,arg2,arg3 ,arg4
```

Where did the quotes go? They were removed when the expanded text was reread by mpp960. To show the difference, try:

```
define('echo1', '$*')
=>
define('echo2', '$@')
=>
define('foo', ``This is macro foo.``)
=>
echo1(foo)
=>This is macro This is macro foo..
echo2(foo)
=>This is macro foo.
```

A `$` sign in the expansion text that is not followed by anything that mpp960 understands is simply copied to the macro expansion, as is any other text.

```
define('foo', '$$$ hello $$$')
=>
foo
=>$$$ hello $$$
```

If you want a macro to expand to a value such as `$12`, put a pair of quotes after the `$`. This prevents mpp960 from interpreting the `$` sign as a reference to an argument.

Deleting a Macro

A macro definition can be removed with `undefine`:

```
undefine('name')
```

which removes the macro `name`. The macro name must be quoted, since it is expanded otherwise.

The expansion of `undefine` is void.

```
foo
=>foo
define('foo', 'expansion text')
=>
foo
=>expansion text
undefine('foo')
=>
foo
=>foo
```

It is not an error for *name* to have no macro definition. In that case, `undefine` does nothing.

Renaming Macros

It is possible to rename an already defined macro with the built-in `defn`:

```
defn('name')
```

which expands to the quoted definition of *name*. If the argument is not a defined macro, the expansion is void.

If *name* is a user-defined macro the quoted definition is simply the quoted expansion text. If *name* is a built-in, the expansion is a special token that points to the built-in's internal definition.

This token is meaningful only as the second argument to `define` (and `pushdef`) and is ignored in any other context. Its normal use is best understood through an example that shows how to rename `undefine` to `zap`:

```
define('zap', defn('undefine'))
=>
zap('undefine')
=>
undefine('zap')
=>undefine(zap)
```

In this way, `defn` can be used to copy macro definitions and definitions of built-in macros. Even if the original macro is removed, the other name can still be used to access the definition.

Temporarily Redefining Macros

It is possible to redefine a macro temporarily, reverting to the previous definition at a later time. This is done with the built-ins `pushdef` and `popdef`:

```
pushdef(`name', `expansion')  
popdef(`name')
```

which are quite analogous to `define` and `undefine`.

These macros work in a stack-like fashion. A macro is temporarily redefined with `pushdef` which replaces an existing definition of `name` while saving the previous definition before the new one is installed. If there is no previous definition, `pushdef` behaves exactly like `define`.

If a macro has several definitions (of which only one is accessible), the topmost definition can be removed with `popdef`. If there is no previous definition, `popdef` does nothing.

If a macro with several definitions is redefined with `define`, the topmost definition is replaced with the new definition. If it is removed with `undefine`, all the definitions are removed, not only the topmost one.

It is possible to temporarily redefine a built-in with `pushdef` and `defn`.

Indirect Call of Macros

Any macro can be called indirectly with `indir`:

```
indir(`name', ...)
```

This results in a call to the macro `name` which is then passed the rest of the arguments. You can use `indir` to call macros with "illegal" names because `define` allows such names to be defined.

(Some macro packages have private macros that can be called only through the built-in `indir`.)

Indirect Call of Built-Ins

Built-in macros can be called indirectly with `builtin`:

```
builtin('name', ...)
```

This results in a call to the built-in `name` which is then passed the rest of the arguments. This can be used if `name` has been given another definition that has covered the original.

Conditionals, Loops and Recursion

mpp960 macros can contain tests and other elements that cause them to evaluate differently at run time.

Testing Macro Definitions

There are two different built-in conditionals in mpp960. The first is

`ifdef`:

```
ifdef('name', 'string-1', ['string-2'])
```

which makes it possible to test whether a macro is defined or not. If `name` is defined as a macro, `ifdef` expands to `string-1`; otherwise to `string-2`. If `string-2` is omitted, it is taken to be the empty string (according to the normal rules).

```
ifdef('foo', 'foo is defined', 'foo is not defined')
=>foo is not defined
define('foo', '')
=>
ifdef('foo', 'foo is defined', 'foo is not defined')
=>foo is defined
```

Comparing Strings

The `ifelse` conditional is much more powerful than `ifdef`. You can use `ifelse` as a way to introduce a long comment, as an if-else construct, or as a multibranch, depending on the number of arguments supplied:

```
ifelse('comment')
ifelse('string-1', 'string-2', 'equal', ['not-equal'])
ifelse('string-1', 'string-2', 'equal', ...)
```

When `ifelse` is used with only one argument, it discards the argument and produces no output. This is a common `mpp960` idiom for introducing a block comment, as an alternative to repeatedly using `dn1`. This special usage is recognized, so that in this case the warning about missing arguments is never triggered.

If called with three or four arguments, `ifelse` expands into `equal` (if `string-1` and `string-2` are equal character for character), otherwise it expands to `not-equal`.

```
ifelse(foo, bar, 'true')
=>
ifelse(foo, foo, 'true')
=>>true
ifelse(foo, bar, 'true', 'false')
=>>false
ifelse(foo, foo, 'true', 'false')
=>>true
```

However, `ifelse` can take more than four arguments. If given more than four arguments, `ifelse` works like a case or switch statement in traditional programming languages. If `string-1` and `string-2` are equal, `ifelse` expands into `equal`, otherwise the procedure discards the first three arguments discarded and repeats. For example:

```
ifelse(foo, bar, 'third', gnu, gnats, 'sixth', 'seventh')
=>seventh
```

A common use of `ifelse` is in macros implementing loops of various kinds.

Loops and Recursion

There is no direct support for loops in mpp960, but macros can be recursive. There is no limit on the number of recursion levels, other than those enforced by your hardware and operating system.

Loops can be programmed using recursion and the conditionals described previously.

The built-in macro `shift` can iterate through the actual arguments to a macro:

```
shift(...)
```

It takes any number of arguments and expands to all but the first argument, separated by commas, with each argument quoted.

How to Debug Macros and Input

Macro debugging in mpp960 is described below.

Displaying Macro Definitions

The built-in `dumpdef` shows what a name expands into:

```
dumpdef(...)
```

This macro accepts any number of arguments. If called without any arguments, it displays the definitions of all known names; otherwise it displays the definitions of the names given. The output is printed directly on the standard error output.

The expansion of `dumpdef` is void.

```
define('foo', 'Hello world.')
=>
dumpdef('foo')
=>foo: Hello world.
=>
dumpdef('define')
=>define: <define>
=>
```

The last example shows how built-in macro definitions are displayed.

Tracing Macro Calls

It is possible to trace macro calls and expansions using the built-ins `traceon` and `traceoff`:

```
traceon(...)  
traceoff(...)
```

When called without any arguments, `traceon` and `traceoff` enables or disables tracing, respectively, for all defined macros. When called with arguments, only the named macros are affected.

The expansion of `traceon` and `traceoff` is void.

The call is displayed whenever a traced macro is called and the arguments have been collected. The expansion can be displayed after the call if the expansion of the macro call is not void. The output is printed directly on the standard error output.

```
define('foo', 'Hello World.')
```

=>

```
define('echo', '$@')
```

=>

```
traceon('foo', 'echo')
```

=>

```
foo  
error-->mpptrace: -1- foo  
=>Hello World.
```

```
echo(gnus, and gnats)
```

error-->mpptrace: -1- echo
=>gnus,and gnats

The number between dashes is the depth of the expansion. The depth is 1 most of the time, signifying an expansion at the outermost level, but it increases when macro arguments contain unquoted macro calls.

See the `d` option (next topic) for information on controlling the details of the debug display.

Controlling Debugging Output

The `d` option to `mpp960` controls the amount of detail presented when using the macros described in the preceding sections.

The *flags* following the `d` option can be one or more of the following:

- a Show the actual arguments in each macro call. This applies to all macro calls if the `t` flag is used, otherwise only the macros covered by calls of `traceon`.
- c Show several trace lines for each macro call. A line is shown when the macro is seen, but before the arguments are collected; a second line is shown when the arguments have been collected, and a third line is shown after the call is complete.
- e Show the expansion of each macro call if it is not void. This applies to all macro calls if the `t` flag is used; otherwise it applies only to the macros covered by calls of `traceon`.
- f Show the name of the current input file in each trace output line.
- i Print a message each time the current input file is changed, giving filename and input line number.
- l Show the current input line number in each trace output line.
- p Print a message when a named file is found through the path search mechanism, giving the actual filename used.
- q Quote actual arguments and macro expansions in the display with the current quotes.
- t Trace all macro calls made in this invocation of `mpp960`.
- x Add a unique macro call id to each line of the trace output. This is useful in connection with the `c` flag above.
- v A shorthand for all of the above flags.

The default is `aeq` if no flags are specified with the `d` option. The examples in the previous two sections assumed the default flags. The built-in macro `debugmode` allows on-the-fly control of the debugging output format:

```
debugmode([ flags ])
```

The *flags* argument should be a subset of the letters listed above. There are three special cases:

1. If the argument starts with a +, the flags are added to the current debug flags.
2. If the argument starts with a -, the flags are removed.
3. If no argument is present, the debugging flags are set to zero (as if `-d` was not given), and with an empty argument the flags are reset to the default.

Saving Debugging Output

Debug and tracing output can be redirected to files using either the `o` option to `mpp960`, or with the built-in macro `debugfile`.

```
debugfile([filename])
```

sends all further debug and trace output to *filename*. If *filename* is empty, debug and trace output are discarded. If `debugfile` is called without any arguments, debug and trace output are sent to the standard error output.

Input Control

This section describes various built-in macros for controlling the input to `mpp960`.

Deleting Whitespace in Input

The built-in `dn1` reads and discards all characters up to and including the first newline:

```
dn1
```

It is often used in connection with `define` to remove the newline that follows the call to `define`. Thus:

```
define('foo', 'Macro 'foo'.')dn1 A very simple macro, indeed.  
foo  
=>Macro foo.
```

The input up to and including the next newline is discarded.

Usually, `dn1` is immediately followed by an end of line or some other white space. `mpp960` produces a warning diagnostic if `dn1` is followed by an open parenthesis. In this case, `dn1` collects and processes all arguments, looking for a matching close parenthesis. All predictable side effects resulting from this collection take place. `dn1` returns no output. The input following the matching close parenthesis up to and including the next newline, on whatever line containing it, is still discarded.

Changing the Quote Characters

The default quote delimiters can be changed with the built-in `changequote`:

```
changequote([start], [end])
```

where `start` is the new start-quote delimiter and `end` is the new end-quote delimiter. If any of the arguments are missing, the default quotes ``` and `'` are used instead of the void arguments.

The expansion of `changequote` is void.

In this example, the `[` and `]` characters are the new quote characters:

```
changequote([,])
=>
define([foo], [Macro [foo].])
=>
foo
=>Macro foo.
```

If no single character is appropriate, *start* and *end* can be of any length.

```
changequote([,])
=>
define([[foo]], [[Macro [[[foo]]].]])
=>
foo
=>Macro [foo].
```

Changing the quotes to the empty strings effectively disables the quoting mechanism, leaving no way to quote text.

```
define('foo', 'Macro `FOO`.')
=>
changequote(,)
=>
foo
=>Macro `FOO`.
`foo`
=>Macro `FOO`.
```

There is no way in *mpp960* to quote a string containing an unmatched left quote, except using `changequote` to change the current quotes.

Neither quote string should start with a letter or `_` (underscore), as they are confused with names in the input. Doing so disables the quoting mechanism.

Changing Comment Delimiters

The default comment delimiters can be changed with the built-in macro `changecom`:

```
changecom([start], [end])
```

where *start* is the new start-comment delimiter and *end* is the new end-comment delimiter. If any of the arguments are void, the default comment delimiters (`#` and newline) are used instead of the void arguments. The comment delimiters can be of any length.

The expansion of `changecom` is void.

Comments are copied to the output, much as if they were quoted strings. If you want the text inside a comment expanded, quote the start comment delimiter.

Calling `changeocom` without any arguments disables the commenting mechanism completely.

Saving Input

It is possible to save some text until the end of the normal input has been seen. Text can be saved to be read again by `mpp960` when the normal input has been exhausted. This feature is normally used to initiate cleanup actions before normal exit, as when deleting temporary files.

Use the built-in `mppwrap` to save input text:

```
mppwrap(string, ...)
```

This stores `string` and the rest of the arguments to be reread when end of input is reached.

The saved input is reread only when the end of normal input is seen, but not if `mppexit` is used to exit `mpp960`.

It is safe to call `mppwrap` from saved text, but then the order in which the saved text is reread is undefined. If `mppwrap` is not used recursively, the saved pieces of text are reread in the opposite order in which they were saved (LIFO — last in, first out).

File Inclusion

`mpp960` allows you to include named files at any point in the input.

Including Named Files

There are two built-in macros in `mpp960` for including files:

```
include(filename)
```

```
sinclude(filename)
```

Both of these cause the file named *filename* to be read by *mpp960*. When the end of the file is reached, input is resumed from the previous input file. The expansion of `include` and `sinclude` is therefore the contents of *filename*.

It is an error for an included file not to exist. If you don't want error messages about non-existent files, use `sinclude` to include a file, if it exists. It expands to nothing if it does not exist.

Normally, file inclusion is used to insert the contents of a file into the input stream. The fact that `include` and `sinclude` expand to the contents of the file can be used to define macros that operate on entire files. The use of `include` is important, as files can contain quotes, commas and parentheses that can interfere with the way the *mpp960* parser works.

Searching for Include Files

mpp960 allows included files to be found in directories other than the current working directory. If a file is not found in the current working directory, and the filename is not absolute, *mpp960* searches for the file in a specified search path.

The directories specified with the `I` option are searched first, in the order found on the command line.

If the `I960INC` environment variable is set, it is expected to contain a colon-separated list of directories, which are searched in order.

If the automatic search for include-files causes trouble, the `p` debug flag can help isolate the problem.

Diverting and Undiverting Output

Diversions are a way of temporarily saving output. The output of *mpp960* can at any time be diverted to a temporary file, and can be reinserted into the output stream later, undiverted.

mpp960 supports up to ten numbered diversions (numbered from 0 to 9). Diversion number 0 is the normal output stream. The number of available diversions can be increased with the `N` option.

Diverting Output

Use `divert` to divert output:

```
divert([number])
```

where *number* is the diversion to be used. If *number* is left out, it is assumed to be zero.

The expansion of `divert` is void.

Diverted output that has not been explicitly undiverted is undiverted when all the input has been processed.

```
divert(1)
```

```
This text is diverted.
```

```
divert
```

```
=>This text is not diverted.
```

```
=>This text is not diverted.
```

```
^D
```

```
=>
```

```
=>This text is diverted.
```

Several calls of `divert` with the same argument do not overwrite the previous diverted text, but append to it.

Output diverted to a non-existent diversion is discarded. This can be used to suppress unwanted output. A common example of unwanted output is the trailing newlines after macro definitions. Here is how to avoid them:

```
divert(-1)
```

```
define(foo, Macro foo.)
```

```
define(bar, Macro bar.)
```

```
divert
```

```
=>
```

This is a common programming idiom in m4.

Undiverting Output

Diverted text can be undiverted explicitly using the built-in `undivert`:

```
undivert([number], ...)
```

which undiverts the diversions given by the arguments in the order given. If no arguments are supplied, all diversions are undiverted in numerical order.

The expansion of `undivert` is void.

```
divert(1)
```

```
This text is diverted.
```

```
divert
```

```
=>This text is not diverted.
```

```
=>This text is not diverted.
```

```
undivert(1)
```

```
=>
```

```
=>This text is diverted.
```

```
=>
```

Notice the last two blank lines. One of them comes from the newline following `undivert`, the other from the newline that followed the `divert`! A diversion often starts with a blank line like this.

When diverted text is undiverted it is not reread; it is copied directly to the current output, and it is therefore not an error to undivert into a diversion.

When a diversion has been undiverted, the diverted text is discarded, and it is not possible to bring back diverted text more than once.

Attempts to undivert the current diversion are silently ignored. `mpp960` allows named files to be undiverted. Given a non-numeric argument, the contents of the file named are copied, uninterpreted, to the current output. This complements the built-in `include`. To illustrate the difference, assume the file `foo` contains the word `bar`:

Diversion Numbers

The built-in `divnum` expands to the number of the current diversion.

Discarding Diverted Text

Often it is not known when output is diverted whether the diverted text is actually needed. A method of discarding a diversion is needed because all non-empty diversions are brought back when the end of input is seen. If all diversions should be discarded, the easiest way is to end the input to `mpp960` with `divert(-1)`.

No output is produced.

Macros for Text Handling

There are built-in macros for manipulating text in various ways, extracting substrings, searching, substituting, and so on.

Calculating Length Of Strings

The length of a string can be calculated by `len`:

```
len(string)
```

which expands to the length of string, as a decimal number.

Searching For Substrings

Use `index` to search for substrings.

```
index(string, substring)
```

expands to the index of the first occurrence of *substring* in *string*. The first character in *string* has index 0. If *substring* does not occur in *string*, `index` expands to -1.

Searching for Regular Expressions

Use the built-in `regexp` to search for regular expressions:

```
regexp(string, regexp, [replacement])
```

which searches for *regexp* in *string*. (The syntax for regular expressions is the same as in GNU Emacs.)

If *replacement* is omitted, `regexp` expands to the index of the first match of *regexp* in *string*. If *regexp* does not match anywhere in *string*, it expands to -1. For example:

```
regexp(GNUs not UNIX, \<[a-z]\w+)
```

```
=>5
```

```
regexp(GNUs not UNIX, \

```

```
=>-1
```

If *replacement* is supplied, `regexp` changes the expansion to this argument, with `&` substituted by *string*, and `\N` substituted by the text matched by the *N*th parenthesized sub-expression of *regexp*, `\0` being the text the entire regular expression matched.

```
regexp(GNUs not UNIX, \w\(\w+\)$, *** \0 *** \1 ***)
```

```
=>*** UNIX *** nix ***
```

Extracting Substrings

Use `substr` to extract substrings.

```
substr(string, from, [length])
```

expands to the substring of *string* which starts at index *from* and extends for *length* characters, or to the end of *string*, if *length* is omitted. The starting index of a string is always 0.

Translating Characters

Character translation is done with `translit`.

```
translit(string, chars, replacement)
```

expands to *string*, with each character that occurs in *chars* translated into the character from *replacement* with the same index.

If *replacement* is shorter than *chars*, the excess characters are deleted from the expansion. If *replacement* is omitted, all characters in *string* that are present in *chars* are deleted from the expansion.

Both *chars* and *replacement* can contain character-ranges such as `a-z` (meaning all lowercase letters) or `0-9` (meaning all digits). To include a dash - in *chars* or *replacement*, place it first or last.

It is not an error for the last character in the range to be larger than the first. In that case, the range runs backwards: `9-0` indicates the string `9876543210`.

Substituting Text by Regular Expression

Global substitution in a string is done by `patsubst`:

```
patsubst(string, regexp, [replacement])
```

This searches *string* for matches of *regexp* and substitutes *replacement* for each match. (The syntax for regular expressions is the same as in GNU Emacs.)

The parts of *string* that are not covered by any match of *regexp* are copied to the expansion. Whenever a match is found, the search proceeds from the end of the match so a character from *string* is never substituted twice. If *regexp* matches a string of zero length, the start position for the search is incremented, to avoid infinite loops.

To make a replacement:

1. Insert *replacement* into the expansion.
2. Substitute *string* for `\&`.
3. Substitute the text matched by the *N*th parenthesized sub-expression of *regexp* (`\0` being the text the entire regular expression matched) for `\N`.

The *replacement* argument can be omitted, in which case the text matched by *regexp* is deleted.

Formatted Output

Use `format` to format output.

```
format(format-string, ...)
```

works much like the C function `printf`. The first argument is a format string that can contain `%` specifications and the expansion of `format` is the formatted string.

The built-in `format` is modeled after the ANSI C `printf` function. It supports the normal `%` specifiers `c`, `s`, `d`, `o`, `x`, `X`, `u`, `e`, `E` and `f`; It also supports field widths and precisions and the modifiers `+`, `-`, `0`, `#`, `h` and `l`. For more details on `printf`, see *C: A Reference Manual*.

Macros for Doing Arithmetic

Integer arithmetic using a C-like syntax is included. There are built-in macros for simple increment and decrement operations.

Decrement and Increment Operators

The built-ins `incr` and `decr` support the increment and decrement of integers:

```
incr(number)
```

```
decr(number)
```

which expand to the numerical value of *number* incremented or decremented, respectively, by one.

Evaluating Integer Expressions

Use `eval` to evaluate integer expressions:

```
eval(expression, radix, [width])
```

expands to the value of *expression*.

Expressions can contain the following operators, listed in order of decreasing precedence.

-	Unary minus
**	Exponentiation
* / %	Multiplication, division and modulo
+ -	Addition and subtraction
== != > >= < <=	Relational operators
!	Logical negation
&	Bitwise and
	Bitwise or
&&	Logical and
	Logical or

\wedge Bitwise exclusive or

All operators, except exponentiation, are left associative. Numbers can be given in decimal, octal (starting with 0), or hexadecimal (starting with 0x). Parentheses may be used to group subexpressions whenever needed. For the relational operators, a true relation returns 1, and a false relation returns 0.

`eval` does not handle macro names, even if they expand to a valid expression or part of a valid expression. All macros must be expanded before they are passed to `eval`.

If *radix* is specified, it specifies the radix to be used in the expansion. The default radix is 10. The result of `eval` is always taken to be signed. The *width* argument specifies a minimum output width. The result is zero-padded to extend the expansion to the requested width.

Note that *radix* cannot be larger than 36 in the current implementation. Any radix larger than 36 is rejected.

Running Host Commands

This section describes the *mpp960* macros that let you run host system commands from within *mpp960*.

Executing Simple Commands

Use `syscmd` to execute any shell command:

```
syscmd(shell-command)
```

executes *shell-command* as a shell command. The expansion of `syscmd` is void.

The expansion is not the output from the command! Instead the standard input, output and error of the command are the same as those of *mpp960*. This means that output or error messages from the commands are not read by *mpp960* and might get mixed with the normal output from *mpp960*,

producing unexpected results. It is therefore a good habit to always redirect the input and output of shell commands used with `syscmd`.

Reading the Output of UNIX Commands

Use `esyscmd` if you want `mpp960` to read the output of a UNIX command:

```
esyscmd(shell-command)
```

This expands to the standard output of the shell command.

The error output of `shell-command` is not a part of the expansion. It appears along with the error output of `mpp960`.

Note that the expansion of `esyscmd` has a trailing newline.

This is not available on Windows hosts.

Exit Codes

Use `sysval` to see whether a shell command succeeded:

```
sysval
```

This expands to the exit status of the last shell command run with `syscmd` or `esyscmd`.

This is not available on Windows hosts.

Making Names for Temporary Files

Commands specified to `syscmd` or `esyscmd` might need a temporary file for output or for some other purpose. Use the built-in macro `maketemp` to make temporary filenames.

```
maketemp(template)
```

This expands to a name of a non-existent file made from the string `template`, which should end with the string `xxxxxx`. The six `xs` are then replaced, usually with something that includes the process ID of the `mpp960` process, in order to make the filename unique.

Several calls of `maketemp` might expand to the same string, since the selection criteria is whether the file exists or not. If a file has not been created before the next call, the two macro calls might expand to the same name.

Printing Error Messages

You can print error messages using `errprint`:

```
errprint(message, ...)
```

which simply prints *message* and the rest of the arguments on the standard error output. The expansion of `errprint` is void.

```
errprint('Illegal arguments to forloop  
' )
```

```
error-->Illegal arguments to forloop
```

```
=>
```

A trailing newline is not printed automatically, so it must be supplied as part of the argument, as in the example.

Two utility built-ins make it possible to specify the location of the error.

```
__file__
```

```
__line__
```

expand to the quoted name of the current input file and the current input line number in that file.

```
errprint('mpp960:'__file__:'__line__': 'Input error  
' )
```

```
error-->mpp960:56.errprint:2: Input error
```

```
=>
```

Exiting from mpp960

If you need to exit from mpp960 before the entire input has been read, use `mpexit`:

```
mpexit([code])
```

This causes mpp960 to exit, with exit code `code`. If `code` is left out, the exit code is zero.

```
define('fatal_error', 'errprint('mpp960:' __file__: __line__':
    fatal error: $*')mpexit(1)')
=>
fatal_error('This is a BAD one, buster')
error-->mpp960: 57.mppexit: 5: fatal error: This is a BAD one,
    buster
```

After this macro call, mpp960 exits with exit code 1. This macro is only intended for error exits, since the normal exit procedures are not followed, e.g., diverted text is not undiverted, and saved text (see `mpwrap`) is not reread.

Compatibility with Other Macro Processors

This section describes the differences between mpp960 and the UNIX System V, Release 3, m4 macro processor.

Extensions in mpp960

mpp960 contains a some facilities that do not exist in UNIX System V m4. These extra facilities are all suppressed by using the `G` option, unless overridden by other command line options.

- In the `$N` notation for macro arguments, `N` can contain several digits, while UNIX System V m4 accepts one digit only. This allows mpp960 macros to take any number of arguments, not only nine.

- When files included with `include` and `sinclude` are not found in the working directory they are sought in a user-specified search path. The search path is specified by the `I` option and the `I960INC` environment variable.
- Arguments to `undivert` can be non-numeric, in which case the named file is included uninterpreted in the output.
- Formatted output is supported through the `format` built-in which is modeled after the C library function `printf`.
- Searches and text substitution through regular expressions are supported by `regexp` and `patsubst`.
- On UNIX (but not in Windows), the output of shell commands can be read into `mpp960` with `esyscmd`.
- There is indirect access to any built-in macro with `builtin`.
- Macros can be called indirectly through `indir`.
- The name of the current input file and the current input line number are accessible through the built-ins `__file__` and `__line__`.
- The format of the output from `dumpdef` and macro tracing can be controlled with `debugmode`.
- The destination of trace and debug output can be controlled with `debugfile`.

In addition to the above extensions, `mpp960` implements the following command line options: `v`, `d`, `l`, `o`, `N`, and `t`. For a description of these options, see the *Invoking mpp960* section.

Also, the debugging and tracing facilities in `mpp960` are much more extensive than in most other versions of `m4`.

Facilities in UNIX System V m4 not in mpp960

There are a few incompatibilities between mpp960 and the UNIX System V m4 tool:

- UNIX System V m4 supports multiple arguments to `defn`. This is not implemented in mpp960.
- When text is being diverted mpp960 implements sync lines differently from UNIX System V m4. mpp960 outputs the sync lines when the text is being diverted, and UNIX System V m4 outputs it when the diverted text is being brought back.
- The problem is determining which lines and filenames should be attached to text that is being, or has been, diverted. UNIX System V m4 regards all the diverted text as being generated by the source line containing the `undivert` call, whereas mpp960 regards the diverted text as being generated at the time it is diverted.
- Invoking mpp960 without the `G` option defines the macro `__gnu__` to expand to the empty string.
- On UNIX systems, mpp960 without the `G` option defines the macro `__unix__`; otherwise the macro `unix`. Both expand to the empty string.

Munger (gmung960)

The gmung960 utility modifies text section and/or data section memory load addresses in an object file. Use gmung960 to load text and/or data at an address other than where it was linked. For example, some code must copy its data from ROM to RAM before execution. In this case, the data is linked at the RAM address, but it must be loaded at the ROM address from which it will be copied. The file's data load address corresponds to the RAM address at link time. After link time, you can modify the data load address to correspond to the ROM address. This lets the ROM burner or loader know the correct address to place the section's contents.

Invoke the munger as:

```
gmung960 [ -option ]... file
```

option is one of the options listed in Table 9-1.

file identifies the object file to be munged. It must be a linked, executable file.

Table 9-1 gmung960 Options

Option	Effect
D [<i>addr</i>]	Changes the load address of the file's data section to <i>addr</i> . <i>addr</i> is interpreted as a decimal, unless preceded by 0x (hex indicator). If <i>addr</i> is omitted, the load address is the first available address following a previously-specified address.
h	gives a help message.
T [<i>addr</i>]	changes the load address of file's text section to <i>addr</i> . <i>addr</i> is interpreted as a decimal number, unless preceded by 0x (hex indicator). If <i>addr</i> is omitted, the load address is either zero or the first available address following a previously-specified address.
v960	writes gmung960 version information to <code>stdout</code> and exits without doing anything.



NOTE. *The section specification options T and D are processed in the order they appear in the invocation.*

Name Lister (*gnm960*, *nam960*)

To display symbolic information on `stdout`, invoke the name lister for:

- relocatable object files
- non-relocatable object files
- libraries
- library members

Unless you specify otherwise, the symbols appear in the order encountered.

With release 6.0, the name lister supports C++. When listing the names of symbols from object files generated by the C++ compiler the name lister displays the demangled name. The demangling function can be disabled by using the `-M` option when invoking the name lister.



NOTE. *Before using the name lister, make sure your object file or archive is in host-endian byte order. To determine byte order, use `gdmp960/dmp960` described in Chapter 6. To change the byte order, use the `cof960/objcopy` converter described in Chapter 3.*

Invoke the name lister as:

```
{nam960}
{gnm960} [-option]... [filename...]
```

<code>nam960</code>	invokes the name lister for backwards compatibility with CTOOLS960 Release 3.5 and later.
<code>gnm960</code>	invokes the name lister for backwards compatibility with GNU/960 Release 2.1 and later.

10

option is any option listed in Table 10-1.

filename specifies the name(s) of one or more files (separated with spaces), whose symbol tables you want the name lister to display. If you do not specify a filename, the name lister tries a.out. You can specify complete pathnames. On UNIX, case is significant in filenames and pathnames.

The symbolic information appears on `stdout`. The display includes:

Code	the section code, as listed in Table 10-2, in lowercase for local symbols and in uppercase otherwise
Name	the symbol name
Value	the symbol value

Table 10-1 gnm960/nam960 Options

Option	Effect
a	displays the debug information.
d	displays the addresses in decimal.
e	displays only the global (external) and static symbols, including the leaf-procedure names.
f	displays all the symbols, including redundant symbols (such as .text, .data, and .bss) that are usually suppressed. This option overrides the <i>g</i> or <i>e</i> option.
g	is the same as <i>e</i> .
h	suppresses the output-header display.
help	displays help information.
M	disables the name demangling function.
n	sorts the symbols alphabetically by name. This option overrides <i>v</i> .
o	displays the addresses in octal.
p	displays the information in a three-column parseable format (the b.out and ELF format default).
R	Reverses the symbol sorting order of the <i>n</i> or <i>v</i> options (sorts in descending order).
r	displays the names of the files defining or referencing each symbol.
s	displays the library symbol map.
T	truncates the symbol names to fit the display-column widths.
u	displays only the undefined symbols. This option overrides <i>g</i> , <i>e</i> , or <i>f</i> .
V	displays the name-lister version and creation date on <code>stdout</code> , and continues processing.
v	sorts the symbols ascending numerically by value.
v960	displays the version and creation date, and stops processing.
x	displays the addresses in hexadecimal (the default).

Table 10-2 Section Codes

Code	Symbol Type
u	undefined symbol
a	absolute (non-relocatable) symbol
t	text-type-section symbol (instructions)
d	data-type-section symbol (initialized data and constants)
b	.bss symbol (uninitialized data)
c	common symbol
o	any other type section symbol (COFF or ELF only)
f	filename symbol
?	debugger symbol-table entry

The COFF display also includes:

Class	a storage class, such as <code>extern</code> for an external symbol or <code>fcn</code> for the beginning and end of a function block
Line	the source line number defining the symbol, for object files containing debug information
Type	the type and derived type, for object files containing debug information
Size	the size in bytes, for object files containing debug information

To suppress the additional COFF information, specify the `p` option.

Symbols are displayed in the order in which they appear in the symbol table, preserving the scoping information. You can sort the symbols by name or address with the `n`, `R`, and `v` options.

Examples

- The following displays the symbols from each of the members of the archive `sample.a`. The name lister displays the filename where each symbol is found. This example uses the `-T` options to truncate symbol names, which keeps the output columns equally spaced.

```
nam960 -r -T sample.a
```

```
Symbols from symbol.a[hello.o]:
```

```
hello.o
```

Name	Value	Class	Type	Size	Line	Section
hell*:gcc2_compiled.	0x00000000	label				.text
h*:__gnu_compiled_c	0x00000000	label				.text
hello.o:_main	0x00000010	extern	()	0x0018		.text
hello.o:_printf	0x00000000	extern				

```
Symbols from symbol.a[byte.o]:
```

```
byte.o
```

Name	Value	Class	Type	Size	Line	Section
bye.o:gcc2_compiled.	0x00000000	label				.text
b*:__gnu_compiled_c	0x00000000	label				.text
bye.o:_main	0x00000010	extern	()	0x0018		.text
bye.o:_printf	0x00000000	extern				

- The following suppresses the header. No column labels appear in the output.

```
nam960 -h hello.o
```

hello.c		file				
_main	0	extern	int()	16		.text
_printf	0	extern				

10

3. The following displays the `proto.o` symbols in parseable format:

```
nam960 -p proto.o
      f proto.c
00000000 T _main
00000352 T _watering
00000368 T _is_time
00000416 T _watered
00000000 U _printf
00000000 U _scanf
00000000 U _init_bentime
00000000 U _exit
00000000 U _bentime
00000000 U _srand48
00000000 U _lrand48
```

4. The following displays only the external symbols:

```
nam960 -e hello.o
Symbols from hello.o:

Name          Value  Class      Type      Size  Line  Section
_main         |      0|extern|  int( )|   16|     |.text
_printf       |      0|extern|           |     |     |
```

5. The following displays the full output:

```
nam960 -f hello.o
Symbols from hello.o:

Name          Value  Class      Type      Size  Line  Section
hello.c       |      | file |           |     |     |
_main         |      0|extern|  int( )|   16|     |.text
.text         |      0|static|           |    2|    3|.text
.data         |     16|static|           |     |     |.data
.bss          |     32|static|           |     |     |.bss
_printf       |      0|extern|           |     |     |
```

6. The following sorts the symbols by name:

```
nam960 -n proto.o
```

```
Symbols from proto.o:
```

Name	Value	Class	Type	Size	Line	Section
_bentime	0	extern				
_exit	0	extern				
_init_bentime	0	extern				
_is_time	368	extern	int()	44		.text
_lrand48	0	extern				
_main	0	extern	int()	348		.text
_printf	0	extern				
_scanf	0	extern				
_srand48	0	extern				
_watered	416	extern	int()	36		.text
_watering	352	extern	arg()	4		.text
proto.c		file				

7. The following sorts the symbols by value:

```
nam960 -v proto.o
```

Name	Value	Class	Type	Size	Line	Section
proto.c		file				
_bentime	0	extern				
_exit	0	extern				
_init_bentime	0	extern				
_lrand48	0	extern				
_main	0	extern	int()	348		.text
_printf	0	extern				
_scanf	0	extern				
_srand48	0	extern				
_watering	352	extern	arg()	4		.text
_is_time	368	extern	int()	44		.text
_watered	416	extern	int()	36		.text

ROM Image Builder (*grom960*)

11

grom960 extracts the text (executable code) and data sections from one or more object files, places them in specified locations in a binary image, and converts the binary image into one or more files in Intel hex format suitable for submission to a PROM programmer. *grom960* also provides options that allow bytes from the binary image to be interleaved into multiple banks of PROMs. *grom960* accepts ELF, COFF, or b.out object file formats as input.

Invocation

The invocation command is:

```
grom960 [ -option ]... section_spec...
```

<i>option</i>	is one of the options listed in Table 11-1. Numeric arguments are interpreted as decimal, unless preceded by 0x (hex).
<i>section_spec</i>	specifies the placement of a text or data section into the binary image. Multiple specifications are allowed; they are processed in the order encountered. There are four types, listed in Table 11-2.

Table 11-1 grom960 Options

Option	Effect
20	generates extended address records in 20-bit format (e.g., as used by the 8086), if the ROM is larger than 64K. The default is to generate 32-bit format records. This option is included primarily for compatibility with old ROM burner software that does not support 32-bit format.
<i>An</i>	sets checksum storage address to <i>n</i> . Default = 0x10000.
<i>bn</i>	generates images for <i>n</i> banks of ROMs. Default = 1.
<i>c 16 32</i>	generates a 16-bit or 32-bit (CRC) checksum.
<i>En</i>	sets checksum end address to <i>n</i> . Default = 0xffff.
<i>f</i>	dumps a full image and does not skip records with all ones.
<i>h</i>	displays help output and exits.
<i>i</i>	suppresses generating hex output files. Instead, dumps the raw binary image to output file image.
<i>l n</i>	generates images for ROMs that are <i>n</i> bytes long. The default is 0x10000 (64K).
<i>m</i>	writes a map of the binary image to stdout.
<i>o name</i>	specifies the base name of the output file(s). When the <i>i</i> option is used, the output file contains the binary image. (Default filename is image.) When the <i>i</i> option is not used, a series of files named <i>namexy.hex</i> contain the hex ROM images. (Default is a series of files named romxy.hex.)
<i>Sn</i>	sets the checksum start address to <i>n</i> . Default is 0x0.
<i>V</i>	prints the version number and continues.
<i>v</i>	produces a map, as with the <i>m</i> option, and summarizes the ROM configuration settings.
<i>v960</i>	writes grom960 version information to stdout and quits.
<i>wn</i>	generates ROMs that are <i>n</i> bytes "wide" (the default is 1). grom960 writes <i>n</i> bytes at a time from the binary image to each bank of ROM, before moving on to the next ROM bank. The combination of the <i>b</i> and <i>w</i> options controls interleaving of ROMs.

Table 11-2 Section Specifications

Section Specification	Effect
<i>filename[,addr]</i>	places the text section of the specified file at address <i>addr</i> , relative to the start of the image, and places the data section immediately following the text section.
B <i>filename[,addr]</i>	also places both the text and data sections of the specified file at address <i>addr</i> , relative to the start of the image. However, the order of the text and data sections is the same as in the input file (i.e., the one linked at the lower address comes first); and any gap between the sections is preserved in the output image.
D <i>filename[,addr]</i>	places the data section of the specified file at address <i>addr</i> .
T <i>filename[,addr]</i>	places the text section of the specified file at address <i>addr</i> .



NOTE. *The `addr` argument is always optional. Omitting the address places the specified section(s) immediately after the one in the preceding specification (or at address 0 in the binary image, in the case of the first section specification).*

Using *grom960*

Generating ROM images is a two-step process:

1. creating a binary image, and
2. converting the image to a ROM image (Intel hex) files.

Creating Binary Images

Regardless of the addresses where the code was linked, all bytes in a ROM image appear in a contiguous address space relative to the ROM's base address. For instance, a 64K ROM based at address 0xffff0000 has a ROM address space of [0,0xffff], byte 0 of the ROM being the byte that is addressed at 0xffff0000 at run time.

The binary image is generated by extracting the text and data sections of the input files and placing them at the specified locations in the ROM address space. Unused address space bytes are initialized to 0xff, the value of a byte in an erased PROM.

Converting the Image to Hex Files

After a single binary image is created, it is interleaved according to the ROM width and the number of banks requested. If the width is w , and the number of banks is b , the first w bytes in the image are written to the first bank of ROMs, the second w bytes are written to the second bank, and so on. After the b th bank has been written, output resumes at the first bank. For example, if the number of banks is four, the ROM width is two, and the first sixteen bytes of the image are:

```
0x00112233445566778899aabbccddeeff
```

then the four banks would begin with the following values:

```
bank 0: 0x00118899...
```

```
bank 1: 0x2233aabb...
```

```
bank 2: 0x4455ccdd...
```

```
bank 3: 0x6677eeff...
```

Each bank corresponds to at least one ROM. Every time the amount of data written to a bank exceeds the specified ROM length, a new ROM image file is started.

Each output file is in Intel hex format and corresponds to a single ROM device. The output files are named *basenamexy.hex*, where y is the bank number and x is the sequence number within the bank. Both x and y are

numbered from 0. For example, if the number of banks is four, the ROM length is 64K, and the total image size is 512K, then the following hex files would be output:

```
bank 0: rom00.hex, rom10.hex
bank 1: rom01.hex, rom11.hex
bank 2: rom02.hex, rom12.hex
bank 3: rom03.hex, rom13.hex
```

Example 1

This example converts the executable `b.out` into ROM images, with text followed immediately by data, with even bytes in one bank and odd bytes in another, for a ROM with 128-Kbyte capacity.

```
grom960 b.out -b 2 -l 0x20000
```

If the binary image is less than 256 Kbytes, there are two output files: `rom00.hex` (even bytes) and `rom01.hex` (odd bytes).

Example 2

This example assumes two `b.out` files as input: `b.out` contains the text and data for an i960® CA processor, and `imi` contains the Initial Memory Image (data that must appear at location `0xfffff00` when the processor powers up). Assume that the total binary image is under 64 Kbytes, and that the ROM will be installed at address `0xffff0000`.

```
grom960 b.out imi,0xff00 -o ca
```

A single output file, named `ca00.hex`, is created.

Example 3

This example makes the same assumptions as Example 2, but data (other than the Initial Memory Image) should appear at location `0x8000` in the ROM (`0xffff8000` in the runtime address space).

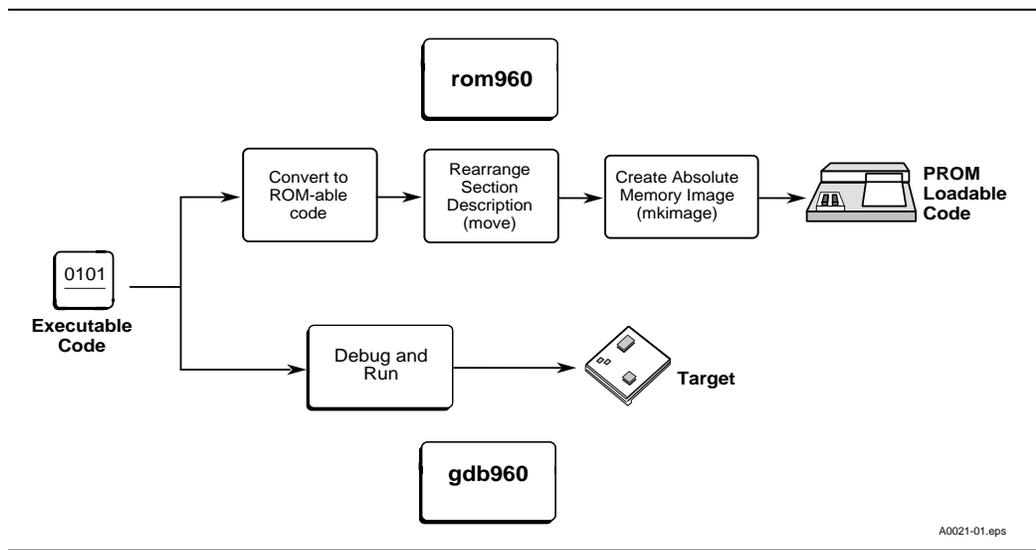
```
grom960 -T b.out -D b.out,0x8000 imi,0xff00 -o ca
```


ROM Image Builder (rom960)

This chapter describes using the rom960 rommer to convert ELF, COFF, or b.out object files to unformatted executable images.

As shown in Figure 12-1, you can prepare code for a specific target environment. The linker generates object files for a downloader. rom960 facilitates rearrangement of section descriptions so that the code can be programmed into programmable read-only memory (PROM) devices.

Figure 12-1 rom960 Rommer Operations



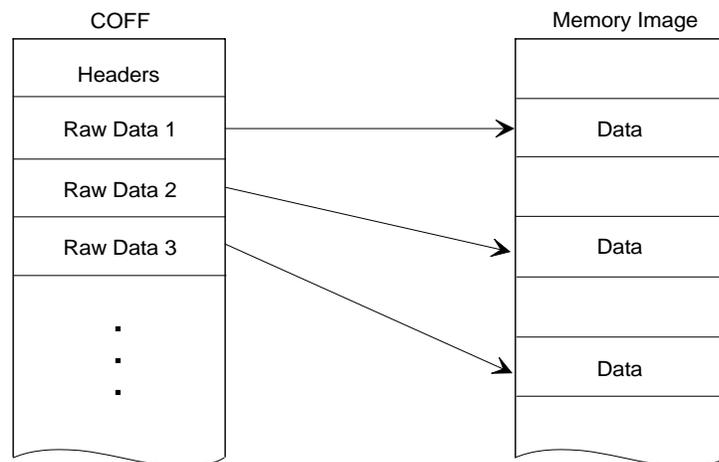
A0021-01.eps

To place code in ROM, code sections must be located at the PROM device addresses. Translating a formatted object file into an unformatted executable image, may include any of the following steps:

- re-ordering the bits to match machine requirements.
- organizing the images to fit the target ROMs.
- calculating a checksum and incorporating it in the image.
- outputting the image in Intel hexadecimal format for a PROM programming device.

Figure 12-2 shows an example of the rommer translation of a COFF file.

Figure 12-2 Data Placement in Memory Image



OSD1702

Rommer Invocation

Use this syntax to invoke the rommer:

```
rom960 -[option] [dfile [arglist]]
```

where *option* is any of the following:

<i>h</i>	displays help.
<i>v</i>	displays the version number and copyright date.
<i>v960</i>	displays version information and exits.
<i>dfile</i>	specifies a rommer directive file named <i>dfile.ld</i> .
<i>arglist</i>	is up to 10 directive-file arguments, separated by spaces.



NOTE. *You must either specify `dfile` in the invocation or supply the directives interactively after entering the `rom960` command. To end an interactive session, type `exit`, `quit`, `Ctrl-z` (in Windows), or `Ctrl-d` (on UNIX) at the rommer prompt.*

Directive Files

The rommer directive filename must have a `.ld` extension. When you run the rommer, you specify the base filename without the `.ld` extension. Table 12-1 lists the rommer directives. For information on using the directives, see the Directive Reference section.

Table 12-1 rom960 Directives

Directive	Operation
<code>checksum</code>	computes and stores a checksum.
<code>help</code>	Displays help information and exits.
<code>ihex</code>	translates an image to an Intel hexadecimal format.
<code>map</code>	reports the section addresses, section sizes, and the image file size to stdout.
<code>mkfill</code>	translates a COFF executable file into a memory image containing an image of the program as it would appear when downloaded.
<code>mkimage</code>	creates an image of the executed file.
<code>move</code>	rearranges the sections.
<code>patch</code>	changes the image contents.
<code>permute</code>	rearranges the address (<code>permute_a</code>) or data (<code>permute_d</code>) bits.
<code>rom</code>	specifies the address space length and width.
<code>sh</code>	executes a host-system command.
<code>split</code>	splits one image into smaller images of the specified size.

You can put rommer directives in your linker-directive files:

- Use only lowercase letters for the rommer directives.
- Start rommer-directive lines with `#*` in columns 1 and 2, putting no space between the `#*` and the rommer directive.
- Separate the rommer directives from the rest of the linker-directive file with `/*` and `*/`. The linker processes no lines between the `/*` and `*/`.

You can write directive files for use with different sets of input. Use the parameters `$0` through `$9` in a directive file to accept arguments sequentially from the rommer invocation. For example, the following uses the `rom.ld` directive file first to build `mypgm.r` from `mypgm.o`, then to build `pgm2.r` from `pgm2.o`:

```
rom960 rom.ld mypgm.o mypgm.s      First invocation
rom960 rom.ld pgm2.o pgm2.s      Second invocation
```

Here is a sample linker-directive file that produces a ROM image:

```
_intr_stack = 0x00040000;
ram_ = 0x30000000;
MEMORY {
    rom: o=0x00000000,l=0x40000
    ram: o=0x301f0000,l=0x40000
}
SECTIONS {
    .text : { } >rom
    .data : { } >ram
    .bss : { } >ram
}
/*
#*move $0
#*mkimage $0 $1
#*split $1 65536 16 65536 8 $1
#*ihex $1.00 $1.eve
#*ihex $1.01 $1.odd
*/
```

The linker processes the assignment statements and the `MEMORY` and `SECTIONS` directives.

The rommer directives appear between the `/*` and `*/` characters and each rommer-directive line starts with `#*` (note that only section headers are changed; no relocation is performed):

<code>move</code>	places the <code>.data</code> section immediately after <code>.text</code> .
<code>mkimage</code>	prepares an image of the data to be programmed into PROMS.
<code>split</code>	divides the image into the two eight-bit-wide units. The first unit contains even-numbered bytes (0, 2, 4 ...); the second contains odd-numbered bytes (1, 3, 5 ...).
<code>ihex</code>	converts the split images into the hexadecimal format required for a PROM programming device.

The `§0` and `§1` characters represent arguments from the rommer invocation.

Directive Reference

This section describes the rommer directives alphabetically. Note that example commands are those that you would place in a linker directive (`.ld`) file or enter at the rommer command prompt.

checksum

Computes and stores a checksum

```
checksum image start-addr end-addr checksum-addr [16 | 32]
```

image is the name of the file in which the checksum is placed.

start-addr is the checksum starting address.

end-addr is the checksum ending address.

checksum-addr is the checksum result address.

[16 | 32] is an option argument to specify a 16- or 32-bit checksum. The default is a 16-bit checksum.

Discussion

To compute a 16- or 32-bit checksum (CRC) over a specified address range in the image file, use `checksum`. The result appears in the image at the `checksum-addr` you specify.

A checksum address beyond the end of the image extends the image, padding any additional intervening bytes with `0xff`.

Example

The following inserts three successive 32-bit checksums in the image `crctest`. The first checksum result is used for the second and the second result is used for the third.

```
rom 32767 8
move hello
mkimage hello crctest
checksum crctest 0 8191 8192 32
checksum crctest 8192 16534 16535 32
checksum crctest 16535 32766 32767 32
```

Related Topic

`mkimage`

ihex

*Translates an image to
an Intel hexadecimal
format*

```
ihex bin hex-file mode
```

bin is the name of the file to be translated.

hex-file is the name of the hexadecimal output file.

mode specifies the hexadecimal address record format (described below).

Discussion

For downloading to an intelligent EPROM programmer, translate a binary image into either of two Intel hexadecimal formats with *ihex*. For the address record format, specify a *mode* as follows:

<code>mode16</code>	specifies a hexadecimal object file format with extended segment address records. This format is used on Intel's 8086 series of 16- or 32-bit processors, and also required by some PROM programmers.
<code>mode32</code>	specifies a hexadecimal object file format with extended linear address records. This format is used on Intel's i960® processors.

Example

The following translates a file named `dhry` into an executable image named `dhryston`, splits the `dhryston` image into two parts, and translates the two parts into new files in hexadecimal format for the PROM programming device:

```
move dhry
mkimage dhry dhryston
split dhryston 65536 16 65536 8 dhryston
ihex dhryston.00 dhryston.even
ihex dhryston.01 dhryston.odd
```

Related Topics

`mkimage`
`split`

map

Reports the section addresses, section sizes, and the image size

```
map infile
```

infile is the filename to be placed in ROM.

Discussion

Use `map` to show how the rommer has restructured the file. The `map` directive displays the following on the `stdout` standard-output device:

- the address and size of each file section
- the size, in bytes, of the projected ROM image

You can use `map` to display information while building a ROM image interactively or from a directive file.

Examples

1. The following shows the `.data` section located in the input file immediately after the `.text` section:

```
##*map a.out

Section name      Physical address      Size
.text            0x04000000            0x5b90
.data            0x040005b90           0x938
.bss             0x60000000            0x118c
Image made from a.out will be 25800 (decimal) bytes
long
```

2. The following shows the `.text` section placed at address 0 and `.data` still at `0x40005b90`. When created with `mkimage`, the space between the sections is filled with zeroes (note the image size):

```
##*move a.out .text 0x0
##*map a.out

Section name      Physical address      Size
.text             0                     0x5b90
.data             0x40005b90           0x938
.bss              0x60000000           0x118c
Image made from a.out will be 1073767624 (decimal)
bytes long
```

3. The following shows `.data` moved to a location that is again immediately after `.text`, producing a smaller image size:

```
##*move a.out
##*map a.out

Section name      Physical address      Size
.text             0                     0x5b90
.data             0x5b90                0x938
.bss              0x60000000           0x118c
Image made from a.out will be 25800 (decimal) bytes
long
```

Related Topic

`move`

mkfill

*Translates a COFF file
into a memory image*

```
mkfill object-input image-output fill-character
```

object-input

image-output

fill-character

Discussion

This command translates a COFF executable file into a memory image containing an image of the program as it would appear when downloaded.

mkimage

*Creates an executable
image*

```
mkimage infile image [section1 [section2 ...]]
```

infile is the object filename.

image is the converted image filename.

section is the name of the section in the *infile* to be converted into the image.

Discussion

This option translates a file into a memory image for downloading or burning into ROMs or PROMs. This directive puts only the raw data of text-type and data-type sections in the image, without the bss-type section, and pads any space between the sections with 0xff.

By default, all text and data sections are translated. However, you may specify sections to translate in the `mkimage` command.

The text-type and data-type section addresses determine the size of the image. The image starts with the lowest-address text-type or data-type section and ends after the highest-address such section.

Examples

The following translates the file `hello` to the binary image `romhello`:

```
mkimage hello romhello
```

The following translates only the three input sections from the file:

```
mkimage file file.image .text .text1 foo
```

The following uses a `mkimage` command after a `rom` command specifying the target ROM configuration.

```
mkimage hello romhello
rom 32767 16 2
```

Related Topic

`move`

move

Rearranges the sections

```
move infile [section { phys-addr | after section } ]
```

infile is the input filename.

section is a linked section name.

phys-addr is the new memory section physical address.

Discussion

This command is used to reorder file sections for programming. `move` changes the address of *section* in the object file to the specified physical address (*phys-addr*). If *section* is not specified, the rommer arranges sections with `.data` immediately after `.text`. When specifying *section*, the physical address may be in hex, or you may use the keyword “after” followed by a section name. For example:

```
move myfile bigblock after .data
```

The `move` directive changes only the header information in *infile*, without relocating any symbols. Use `move` to prepare a file for the `mkimage` directive or (for example) to change the physical address for a particular EPROM.

The `move` command writes the changes to *infile*.



CAUTION. *After the input file has been modified with `move`, the application must contain executable startup code to copy each section that is moved back to its original address, as defined in the linker directive file. Without such executable startup code, the application may execute incorrectly when downloaded, since inter-section references may no longer be valid.*

Example

In the following, the first `move` directive starts the `.text` section at 16383:

```
move hello .text 16383
```

Related Topics

Related Topic

`map`
`mkimage`

packhex

*Compresses a hex file by
repacking the data
records*

```
packhex hex-file
```

hex-file Name of the hex file to repack.

Discussion

This command compresses a hex file by re-packing the data records. The hex is converted in-place. This operation should be done before using a `split` command.

Example

```
packhex myfile
```

patch

Overwrites the image contents with that of a new file

```
patch image infile addr
```

image is the executable binary filename.

infile is the patch filename.

addr is the patch starting address in the binary file.

Discussion

To overwrite part of an executable binary file with the contents of a patch file, use `patch`. Specify the *address* offset in the binary file where the patch is to start. The length of the patch file determines how much of the binary file is overwritten.

Example

The following overwrites code in the `patfile` file with the contents of the `newbyte` file, beginning at address 1000 of `patfile`:

```
move hello
mkimage hello patfile
patch patfile newbyte 1000
```

Related Topic

`mkimage`

permute

Rearranges the data or addresses in a binary file

```
permute-a infile order outfile  
permute-d infile order outfile
```

<i>infile</i>	is the binary filename to be permuted.
<i>order</i>	is a series of integers separated by spaces indicating the order of the bits to be permuted.
<i>outfile</i>	is the name of the file containing the permuted image.

Discussion

Rearrange the address bits or the data bits in a ROM image as follows:

To reorder data bits, use `permute-d`. Bits are repositioned within data items that are the width of the ROM image. Specify the output bit location for each input bit from the least-significant to the most-significant input bit.

To reorder address bits, use `permute-a`. Bits are repositioned within 32-bit addresses. Specify the new bit location for each address bit from least- to most-significant input bit.

In interactive mode, the rommer prompts for the new location of each bit.

Use `permute-a` or `permute-d` after a `rom` command describing the target ROM configuration. The `permute-a` command uses the length and width specifications from the `rom` command to define the working address space.

Examples

1. The following reorders data bits 0 through 7 in `test` and places the new image in `bldrom`:

```
rom 64000 8
move hello
mkimage hello test
permute-d test 0 2 4 6 1 3 5 7 bldrom
```

2. The following reverses address bits 16 through 31, placing the new image in `hello2`:

```
rom 120000 32
move hello
mkimage hello romhello
permute-a romhello 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
hello2
```

Related Topics

mkimage split
rom

rom

*Specifies the image
length and width*

`rom` *length*, *width*, [*count*]

length is the number of bytes in the ROM image.

width is the width of the ROM image, in bits.

count is the maximum number of ROM images.

Discussion

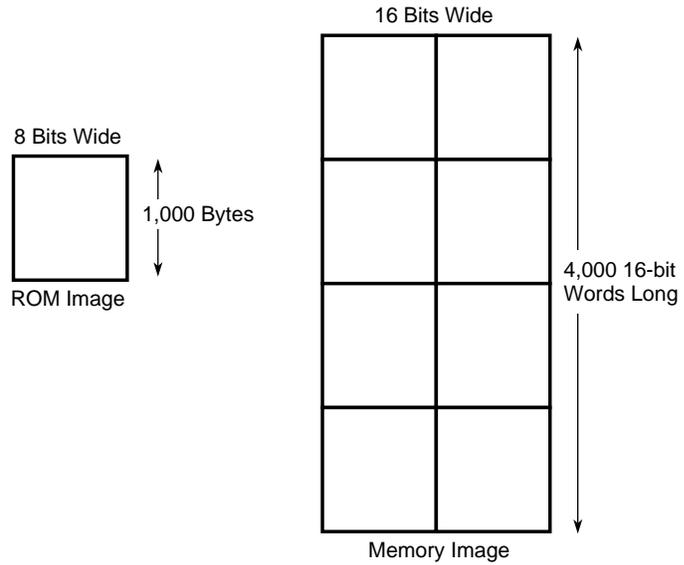
Use this option to specify the length, width, and number of ROM images before using a `permute` command. Figure 12-3 shows an eight-ROM memory image and an example of the command required to specify it.

If you specify a `count`, the rommer issues a warning message for any `mkimage` output greater than the expected memory size (in bytes). To determine the expected memory size, use the formula:

$$\text{length} * \text{count}$$

The `permute-d` command uses `width` and `permute-a` uses both `length` and `width` as the number of bits to be permuted.

Figure 12-3 Dimensions of a Memory Image



ROM command required to use either the `permute-a` or `permute-d` command:

```
rom 1000 8 8
```



Quantity needed to fill the memory image
Description of one ROM

A0123-01

Examples

1. The following sets the length to 32767 bytes and the width to 8 bits for individual ROM images:

```
rom 32767 8
```

2. The following sets the length to 65536 bytes and the width to 8 bits; and, by specifying four ROM images of those dimensions, establishes 262144 bytes as the expected image size:

```
rom 65536 8 4
```

Related Topics

mkimage
permute

sh

*Executes a host-system
command*

sh *command*

command is the host-system command to be executed.

Discussion

To execute a command on your host system, use `sh`. When the command completes, the rommer continues executing.

Examples

1. The following lists the current directory on UNIX:

```
sh ls -l
```
2. The following lists the current directory in Windows:

```
sh dir
```

split

*Specifies the output
image sizes*

```
split image m-length m-width r-length r-width name
```

<i>image</i>	is the binary filename to be split.
<i>m-length</i>	is the length, in bytes, of the memory image to be split.
<i>m-width</i>	is the width, in bits, of the memory image to be split.
<i>r-length</i>	is the ROM image length, in bytes.
<i>r-width</i>	is the ROM image width, in bits.
<i>name</i>	is a base name for the various images produced.

Discussion

To split a memory image into smaller ROM images, use `split`. Each output ROM image filename is of the form *name.nm*:

- n* indicates the vertical position of the ROM image, from 0 (the lowest part of the memory address range) to $(m-length / r-length)$.
- m* indicates the horizontal position of the ROM image, from 0 (the first block of *r-width* bits in the *m-width*-bits-wide data path) to $(m-width / r-width)$.

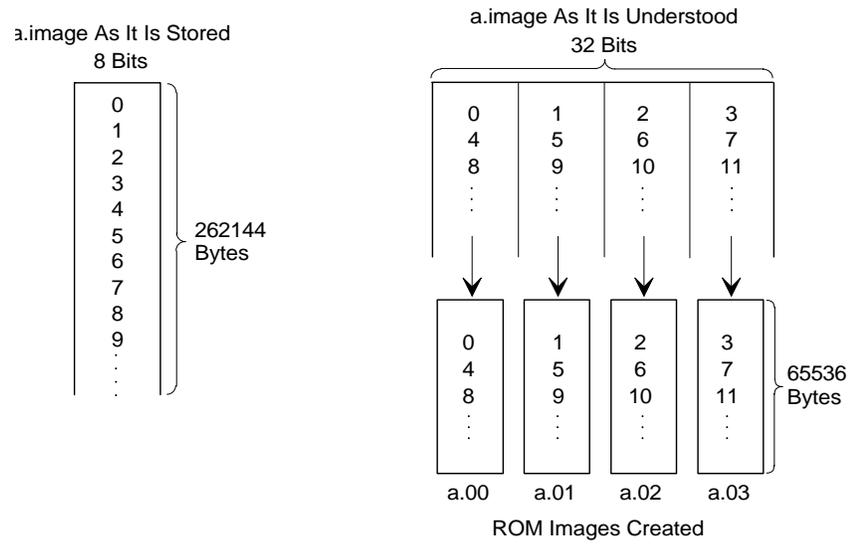
Subsequent `permute` directives use the ROM image *r-length* and *r-width*.

Example

The following produces four files named *a.nm*, as shown in Figure 12-4:

```
mkimage hello a.image
split a.image 262144 32 65536 8 a
```

Figure 12-4 split Command Example



OSD1701

Related Topics

mkimage
permute-d

Section-size Printer

(*gsize960*, *siz960*)

13

This chapter describes the section-size printer, used for displaying the sizes of archive members, object-file sections, and entire files.



NOTE. Before using this tool, make sure your object file or archive is in host-endian byte order. To change the byte order, use the *cof960/objcopy* converter described in Chapter 3.

Invocation

Invoke the section-size printer as:

```
{siz960 } [-options] filenames  
{gsize960 }
```

<i>siz960</i>	invokes the section-size printer for backwards compatibility with CTOOLS960 Release 3.5 and later.
<i>gsize960</i>	invokes the section-size printer for backwards compatibility with GNU/960 Release 2.1 and later.
<i>options</i>	is one or more of the options listed in Table 13-1.
<i>filenames</i>	is one or more filenames, separated with spaces, for which the symbol tables are to be displayed. You can specify complete pathnames.

Table 13-1 gsize960/siz960 Options

Option	Effect
c	displays total size of common symbols.
d	displays the sizes and addresses in decimal.
h	displays help information and exits.
n	Includes unallocated sections in the size calculation.
o	displays the sizes and addresses in octal.
p	Suppresses header display.
V	displays the section-size printer version and creation date and continues processing.
v960	displays the section-size printer version and creation date and stops processing.
x	displays the sizes and addresses in hexadecimal.

By default, sizes are displayed in decimal and addresses in hexadecimal. For example, the following displays the version information and continues processing:

```

siz960 test.o
test.o :
Section          Size      Address

.text            52      0x00000000
.data             4      0x00000034
.bss              0      0x00000038

Total            56

```



NOTE. With release 5.1, the `-n` option includes ELF `.debug` sections in the size calculation. Note that using `siz960/gsize960` with the `-n` option produces output that is identical to that produced by version 5.0 of the sizer.

Statistical Profiler (ghist960)

Overview

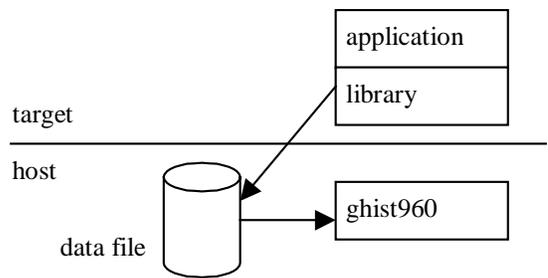
The ghist960 utility facilitates statistical analysis of the execution behavior of programs containing debug information. Ghist960 works with:

- b.out, COFF, and ELF files
- big- or little-endian byte ordered files
- IxWorks and MON960 monitors.

The statistical profiler consists of two separate components:

Component	Runs On	Function
1) ghist960 reporting tool	Host	Displays performance data created by the ghist960 run-time library. After profiling is completed, this data is stored in a file that resides on the host.
2) ghist960 run-time library	Target	The target-specific library that is linked with the code to be analyzed. During profiling, the library collects performance data and writes this data to the host.

Figure 14-1 ghist960 Profiler Environment





NOTE. *Statistical profiling differs from precise profiling, which is described in the i960® Processor Compiler User's Guide.*

Table 14-1 Terminology

Term	Definition
ghist960 library	Target-dependent run-time library (either libhs or libixhs)
libhs	MON960 version of the ghist960 run-time library
libixhs	IxWorks version of the ghist960 run-time library
library variable	A variable defined in the ghist960 library that may be modified by the user (i.e. <code>_buck_size</code>)
host	Hardware that runs CTOOLS and/or Tornado
target	Hardware that runs application under test and/or IxWorks
msec	Milliseconds
<code>_btext</code>	Beginning of the application's text (instruction) section
<code>_etext</code>	End of the application's text (instruction) section
IP	Instruction Pointer
TIMER1	hardware timer 1 (not available on some processors)
TCR1	Timer Count Register 1
TRR1	Timer Reload Register 1
TMR1	Timer Mode Register
IMSK	Interrupt Mask

How Statistical Profiling Works

Statistical profiling uses a sampling technique that records the IP at fixed intervals. The recorded IP is correlated with a hit-counter that contains the number of times the IP has fallen within its range of memory. This range of memory is referred to as a bucket.

When profiling terminates, each counter will contain a value that represents the frequency of occurrence that the IP spent within each counter's bucket.

If enough samples were taken, these values will be statistically significant such that they will accurately show where the IP was most of the time.

In the example shown in Figure 14-2, a simple function is assembled and the resulting assembly occupies exactly 32 bytes. The bucket size for this example is 16 bytes. If the IP resides within the range of 0x0-0x10 during sampling, the hit-counter for bucket0 will be incremented. If the IP is within 0x14-0x20, the hit-counter for bucket1 will be incremented.

Figure 14-2 Buckets

source	assembly	
1 void f1 ()	0: lda 0x20,g14	} → bucket 0
2 {	8: mov g14,g0	
3 int k;	c: mov 0,g14	
4 for (k=0; k<9; k++)	10: mov 8,g4	
5 {	14: subo 1,g4,g4	} → bucket 1
6 }	18: cmpible 0,g4,0x14	
8 }	1c: bx (g0)	
	20: ret	

When profiling is completed, the performance data is transferred to the host so it can be displayed using the ghist960 reporting tool. If in the example above, bucket0 receives 1% of the program's total execution time, and bucket1 receives 5%, ghist960 will report something similar to this:

```

% time  function name  line    file name  address
=====
1.0%   f1              1      ex.c      0x00000000
5.0%   f1              4      ex.c      0x00000014

```

Debugging information embedded within the executable allows ghist960 to report roughly which lines of C source code were executed most frequently.

Parameters that Effect Profiling

The ghist960 library defines several user-modifiable parameters that influence profiling operation. The easiest way to manipulate these parameters is through the use of gld960's `-defsym` command line option, although in certain cases it may be useful to modify `_prof_start` and `_prof_end` in the application's source code. If you use the `-defsym` command line switch, the symbol must be preceded by two underscores. If you are using the symbols directly in source code, they must be preceded by only one underscore.

While usage of library parameters differs slightly depending on which version of the library you are using, all parameters are of type integer and must be positive in value. Refer to tables 14-2 and 14-3 for MON960 and IxWorks specific usage, respectively.

Table 14-2 MON960 Library Parameters (libhs)

Defsym	Default	Possible Values	Effect
<code>__buck_size*</code>	0x40	>0	Number of text space bytes per bucket.
<code>__timer_freq</code>	0x2	0x1 0x2 0x3 0x4	Timer interrupt frequency. Use one of the following settings. (Any other setting resets the default.) 500 mSec 1 mSec 2 mSec 5 mSec
<code>__prof_start*</code>	<code>_btext</code>	<code>>=_btext</code> and <code><_prof_end</code>	The low address in the range of instruction addresses to profile.
<code>__prof_end*</code>	<code>_etext</code>	<code><=_etext</code> and <code>>_prof_start</code>	The high address in the range of instruction addresses to profile.

<code>_heap_size</code>	0x2000 (8KB)	Memory allocated for heap space (MON960). Heap space is where the raw profile data is kept for later dumping to the host. Default is usually insufficient for profiling. To determine the <i>extra</i> heap size you need for profiling, use the formula: $_heap_size += ((__prof_end - __prof_start) / __bucket_size) * sizeof(int)$
-------------------------	-----------------	---



***NOTE.** Changing this parameter directly affects heap space requirements. See `_heap_size` for details.

Table 14-3 IxWorks Library Parameters (libixhs)

Defsym	Default / Typical	Possible Values	Effect
<code>__buck_size*</code>	none / 0x10	2^n , where $n \geq 3$ <0xffff	Number of text space bytes per bucket.
<code>__timer_freq</code>	none / 0x2710	>0	Timer interrupt frequency. Expressed in the number of times to sample in one second.
<code>__prof_start*</code>	none	Must be first symbol in <code>.text</code> section.	The low address in the range of instruction addresses to profile.
<code>__prof_end*</code>	none	Same as MON960	The high address in the range of instruction addresses to profile



NOTE. There are no default values for `libixhs`. All values must be supplied.

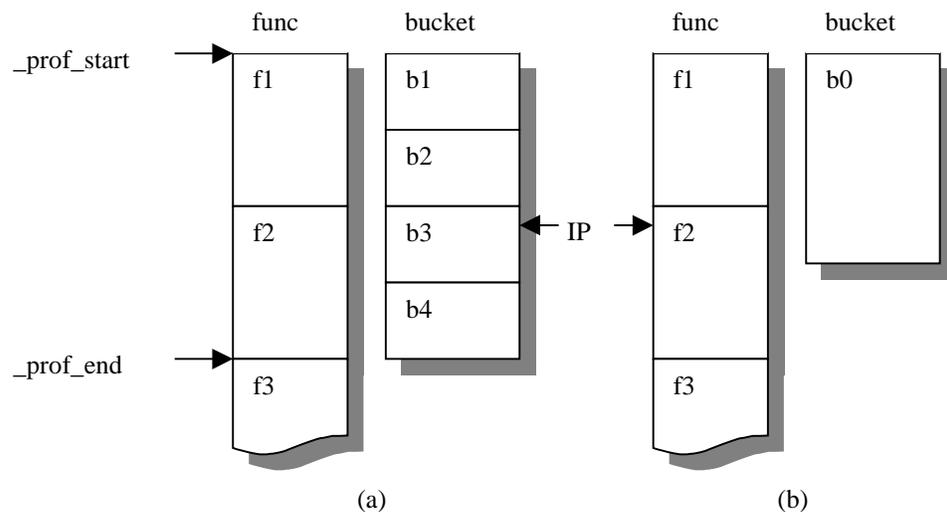
Bucket Size (`__buck_size`)

The bucket size determines the granularity of measurement. A hit-counter is allocated on the application's heap for each bucket, so smaller bucket sizes translate into more heap memory usage but better measuring accuracy.

In reality, a bucket may cross function boundaries as shown in figure 14-3b. Situation (b) occurs more often than situation (a). When the IP shown in figure 14-3b is sampled, the hit-counter for bucket b0 will be incremented. Ghist960 will report a result that makes function f1 look like it has received more IP hits, because the counter is associated with the first line of code.

For libixhs, the bucket size must be a power of 2 and larger than 4.

Figure 14-3 Buckets Crossing Function Boundaries



Timer Frequency (`__timer_freq`)

Sampling is achieved by using a hardware timer. Depending on the hardware configuration, the timer used may either be the i960® processor's TIMER1 or an off-chip peripheral.

The usage of `__timer_freq` depends on which library you are using. With `libhs`, the only valid values are 0x1-0x4. With `libixhs`, the value of `__timer_freq` is the number of times to sample in one second. The range of valid `__timer_freq` values is given by :

$$0 \leq \frac{busClkMhz * 10^6}{_timer_freq} \leq \text{ffff ffff}_{16}$$

Where *busClkMhz* is the speed of the CPU clock in megahertz.



NOTE. *Profiling requires complete control of the target board's timer. If the program being profiled uses the timer, the profile data is incorrect or misleading. You must disable all timer accesses*

Profiling region (`__prof_start` and `__prof_end`)

Most often, the entire program will be profiled. It is also possible to select a region for profiling by modifying `__prof_start` and `__prof_end`. Code that does not reside within the bounds of `__prof_start` and `__prof_end` will not be profiled.

When developing for the MON960 environment (`libhs`), the low-level library (`libll`) defines `__prof_start` and `__prof_end` to `_btext` and `_etext`, respectively. In contrast, when developing for IxWorks, `__prof_start` and `__prof_end` *must* be set explicitly. Furthermore, `libixhs` places special restrictions on `__prof_start`, which are described here in detail.

There are four possible scenarios in which to select code for profiling, as shown in Figure 14-4. Three methods for modifying `__prof_start` and `__prof_end` are presented in table 14-5.

Figure 14-4 Four Scenarios for Selecting Profiling Regions

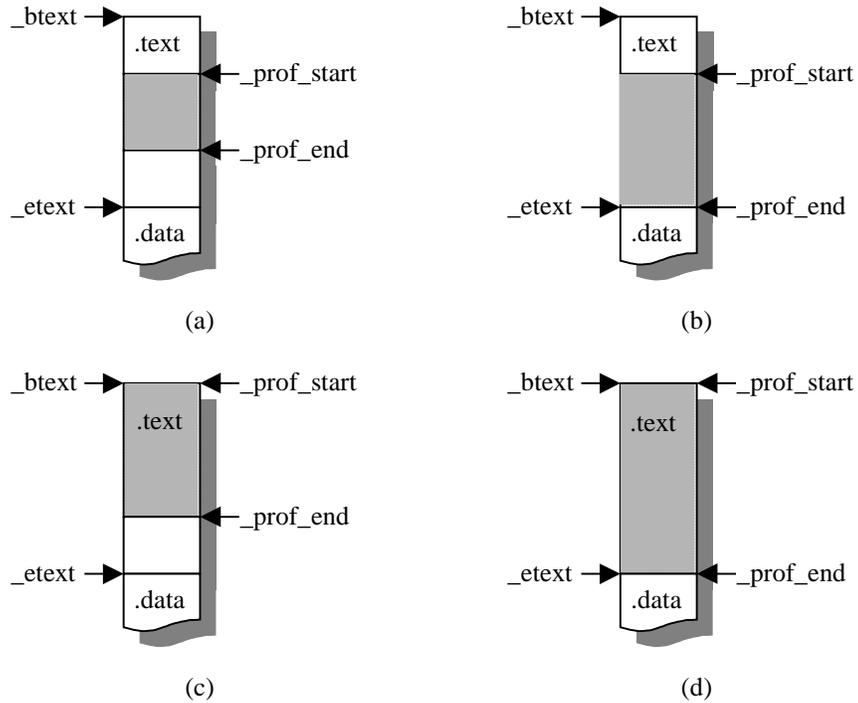


Table 14-4 Preferred Methods for Modifying `_prof_start` and `_prof_end`

Figure	Region to profile	Method		Works with	
		<code>_prof_start</code>	<code>_prof_end</code>	libhs	libixhs
14-4 (a)	Between two selected addresses	1,2	1,2	•	
14-4 (b)	Selected address to end of <code>.text</code> section	1,2	3	•	
14-4 (c)	Beginning of <code>.text</code> section to selected address	3	1,2	•	•
14-4 (d)	Entire <code>.text</code> section	3	3	•	•

Table 14-5 Methods of Modifying `__prof_start` and `__prof_end`

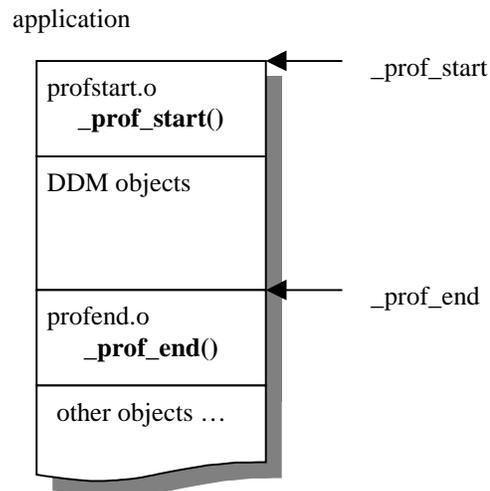
method	description
1	<p>The <code>defsym</code> command line switch can be used to assign values to <code>__prof_start</code> and <code>__prof_end</code>. The <code>-defsym</code> command line switch can be presented to the compiler or linker. Symbols must be preceded with an extra underscore (i.e. <code>__prof_start</code>).</p> <p>In the following example, code will be profiled between the starting addresses of <code>main</code> and up to the starting address of <code>endapp</code> (<code>endapp</code> itself will not be profiled).</p> <pre>gcc960 [...] -defsym __prof_start=_main \ -defsym __prof_end=_endapp</pre>
2	<p><code>_prof_start</code> and <code>_prof_end</code> dummy place-holder functions may be defined in the same file within the application. Code between these dummy place-holders will be profiled.</p> <pre>... void _prof_start() {} <i>(functions to profile)</i> void _prof_end() {} ...</pre>
3	<p>The dummy place-holders <code>_prof_start</code> and <code>_prof_end</code> are defined in their own separate files. The files are positioned around the files containing functions to profile on <code>gld960</code>'s command line, and contain no other code except the dummy place-holders.</p> <p><code>profstart.c</code> contains:</p> <pre>void _prof_start() {}</pre> <p><code>profend.c</code> contains:</p> <pre>void _prof_end() {}</pre> <p>The files are then linked in the following manner:</p> <pre>gld960 [...] profstart.o <i>(profiled objects)</i> profend.o <i>(other objects)</i></pre> <p>This is the recommended approach to guarantee that <code>_prof_start</code> is the first symbol located in the <code>.text</code> section.</p>



NOTE. For *libixhis*, `_prof_start` must be the first symbol placed in the `.text` section.

Figure 14-4 shows how an IxWorks DDM object can be profiled. 'DDM objects' will be profiled because they are between `_prof_start` and `_prof_end`. However, 'other objects' will not be profiled.

Figure 14-4 DDM Memory Layout



Resource Requirements

The library requires the use of one hardware timer and one hardware-generated interrupt. The specific timer and interrupt used are platform and hardware configuration dependent.

Table 14-6 Required Resources

Resource	MON960	IxWorks
Hardware Timer	TIMER1	Rx TIMER1
Interrupt Vector	11	0xe2

Library Initialization

MON960

When developing code for use with MON960, the low-level library (libll) is linked in by default. The start-up code defined in libll automatically calls `_init_profile()` and `_term_profile()`, requiring no intervention from the user.

IxWorks

When developing for IxWorks, the MON960 C low-level library is not linked with your application. In addition, IxWorks uses incremental linking. Therefore, `_init_profile()` and `_term_profile()` must be explicitly called by either the shell or the application to start and end profiling, respectively.

When `_init_profile()` is called, the following sequence occurs:

1. The current interrupt vector for TIMER1 is saved
2. New vector for TIMER1 is programmed
3. TCR1, TRR1, TMR1 are modified.
4. TIMER1 interrupt is enabled (IMSK modified)

When `_term_profile()` is called, the following occurs:

1. The old interrupt vector is restored
2. TMR1 is reset



NOTE. *TCRI, TRRI, and TMR1 are not restored to their original values when `_term_profile()` is called*

Invocation

Invoke the statistical profiler with:

```
ghist960 [option]... program [data_file]
```

<i>option</i>	is one of the options listed in Table 14-1.
<i>program</i>	identifies the application (object file) for which you are creating the report.
<i>data_file</i>	names the execution profile data file, which can be in ASCII ('.asc') or binary ('.dat') form. The file <i>must</i> have an '.asc' or '.dat' extension.

The ghist960 tool processes the object file by:

1. Building an internal database of function entry and line number symbols.
2. Determining data file format type by file's extension.
3. Processing the profile data, matching "bucket" hit counts to their associated symbols and line number entries. A bucket is a range of addresses. The number of address bytes in the bucket (bucket size) determines the range of addresses corresponding to a bucket.
4. Printing the internal database for each symbol/line entry, which includes:
 - the percentage of hits for this particular symbol/address range
 - the function name associated with the symbol
 - the source file line number of the symbol
 - the file where the symbol resides
 - the address of the symbol

Optionally, you can use the `n` option to obtain the actual number of hits in the address range.

Table 14-7 ghist960 Options

Option	Effect
<code>a</code>	prints all buckets with one or more hits. (Normally, only buckets with 1% or more of the hits are printed.)
<code>d</code>	prints ghist960's internal symbol table and the standard data.
<code>f</code>	tallies function information. Counts the number of hits for each line in a function and associates the sum with line zero of the function.
<code>h</code>	displays help output and exits.
<code>n</code>	prints the number of hits for each bucket, in addition to the standard data.
<code>s</code>	suppresses the printing of miscellaneous header and footer information and format lines.
<code>V</code>	writes version information to stdout and continues.
<code>v960</code>	writes version information to stdout and exits.

Using ghist960

Statistical analysis using ghist960 is a multiple step process. Generally, the procedure is slightly different whether you are using the MON960 or the IxWorks library.



NOTE. *When compiling the application to be analyzed, it is best to specify the `-g` (debug) flag for symbolic debugging allowing ghist960 to report line numbers.*

Table 14-8 Procedure for Profiling Under MON960 and IxWorks

Phase	Step	Instruction	MON960	IxWorks
Compile / Link	1	Define profiling region.	<i>optional</i>	<i>required</i> Assign values to _prof_start _prof_end
	2	Compile with -g flag	<i>recommended</i>	<i>recommended</i>
	3	Link with run-time library	<i>required</i> (libhs)	<i>required</i> (libixhs)
	4	Modify library parameters with 'defsym' switch	<i>optional</i>	<i>required</i> Assign values to _timer_freq _buck_size
Run	1	Download app to be profiled to target	<i>required</i>	<i>required</i>
	2	Initialize library	<i>automatic</i>	<i>required</i> Call _init_profile() from IxWorks shell
	3	Run program with normative data	<i>required</i>	<i>required</i>
	4	Terminate profiling	<i>automatic</i>	<i>required</i> Call _term_profile() from IxWorks shell and redirect output to file with '.asc' extension.
Analysis	1	Invoke ghist960	<i>required</i>	<i>required</i>



NOTE. Because the IxWorks version of the library uses *stdio*, Virtual I/O must be enabled by the shell.

Description of File Formats Emitted by the Library

After `_term_profile()` is called, performance data will be sent to the host. If using `libhs`, a binary `.dat` file will automatically be created in the directory `mondb` was invoked in. If using `libixhs`, the data is sent to `stdout`, which must then be manually redirected to a file on the host. It is important that this file has an `.asc` extension, indicating that it is in ASCII form.

Each version of the library has generally the same format:

Figure 14-5 Data File Format

bucket size
IP / hit Pair
IP / hit Pair

...

Depending on which version of the library you are using, the file will be stored in either ASCII or binary format. `Ghist960` distinguishes the file's format by its extension.

Table 14-9 File Formats Generated by the ghist960 Library

Operating System	Format	Extension
lxWorks	ASCII	.asc
MON960	binary	.dat

Binary Data Format

The binary format stores integers in little-endian byte-order. The first integer is the bucket size, followed by alternating sequences of IP addresses and hit counts. The following table describes this format.

Table 14-10 Binary Data Format

Bytes	Declaration	Name	Description
0-3	int	bucket size	The size of each bucket used during profiling. See the library variable '_buck_size' for more information.
4-7	int	IP	First IP value
8-11	int	hit counter	Hit-counter associated with first IP.
12-15	int	IP	Second IP value
16-19	int	hit counter	Hit-counter associated with second IP.
...	int

ASCII Data Format

The ASCII file format stores an IP followed by a hit-count on each line. The IP and hit-counter are separated by a space and followed by a new-line. The exception to this is the first line in the file, which only contains the bucket size. All numbers are represented as hexadecimal, but are not preceded by '0x'.

Table 14-11 ASCII File Format

Line	Contents (separated by a space)
0	bucket size
1	IP0 #hits
2	IP1 #hits
n

Example Usage of ghist960

The following example is a simulation that keeps the IP busy in several functions. The algorithm defined in the function `run_simulation` is designed to call a 'bin' function with a frequency that corresponds to a normal distribution: `bin1` and `bin13` should have relatively few hits whereas `bin7` should get most of the hits. The simulation is run 1000 times.

```
/* example.c */

#define DELAY 5000
#define WAIT { volatile int i; for ( i=0; i<DELAY; delay++ ){} }

void bin1 () { WAIT }
void bin3 () { WAIT }
void bin5 () { WAIT }
void bin7 () { WAIT }
void bin9 () { WAIT }
void bin11 () { WAIT }
void bin13 () { WAIT }

void run_simulation (void)
{
    int level,pos=6;

    for ( level = 1; level <= 5; level++ ) {
        if ( rand() % 2 )
            pos--;
        else
            pos++;
    }

    switch ( pos )
    {
        case 1 : bin1(); break;
        case 3 : bin3(); break;
        case 5 : bin5(); break;
        case 7 : bin7(); break;
        case 9 : bin9(); break;
        case 11 : bin11(); break;
        case 13 : bin13(); break;
    }
}
```

```

void main ( )
{
    int sim;

    for ( sim=0; sim < 1000; sim++ )
        run_simulation();
}

/* End of example.c */

```

For this example, a hypothetical scenario for profiling is created.

Table 14-12 Example Parameters

Library parameter	Value	Explanation
_timer_freq	0x01	Set sampling rate at 500 mSec.
_prof_start	bin1	Start profiling at the address where _bin1 is defined.
_prof_end	run_simulation	end profiling up to the address where run_simulation is defined.

The gcc960 command line will look as follows :

```

gcc960 -AJF -Fcoff -g -Tmcyjx -lhs example.c \
    -defsym __prof_start=_bin1 \
    -defsym __prof_end=_run_simulation \
    -defsym __buck_size=0x1 -o example

```

After example.c has been successfully compiled and linked, it must be run on the host. If using MON960, this could be done as follows :

```
mondb example
```

After the application has completed execution, ghist.dat will reside on the host. To view the results, invoke ghist960:

```
ghist960 -a example ghist.dat
```

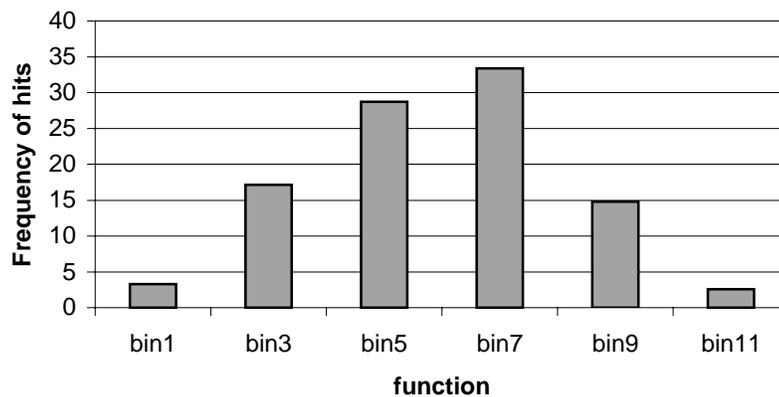
The following output is representative of the profiling session (values may change slightly when example is re-run.)

```
Gcc 80960 History Profiler V3.0

Reading data from 'ghist.dat' Done.

=====
Profile of time spent in program 'example'.
1947 samples taken with bucket size of 64.
33 line numbers found.
=====
% time   function name  line   file name  address
=====
 33.4%      bin7      7     bins.c  0xa00088d0
 28.7%      bin5      6     bins.c  0xa0008890
 17.1%      bin3      5     bins.c  0xa0008850
 14.8%      bin9      8     bins.c  0xa0008910
   3.3%      bin1      4     bins.c  0xa0008810
   2.6%     bin11      9     bins.c  0xa0008950
=====
```

Figure 14-6 Hit Frequency for Bin Functions



Stripper (*gstrip960*, *str960*)

The stripper removes symbol-table and line-number information from your files. In most cases, you will want to use the stripper after debugging to reduce the code size by removing debug symbols.



NOTES.

- The stripper overwrites the input file with the stripped file.
 - You cannot link or symbolically debug a stripped file.
-

Invoke the stripper as:

```
{str960
  gstrip960} [-option]... filename...
```

<code>str960</code>	invokes the stripper for backwards compatibility with CTOOLS960 Release 3.5 and later.
<code>gstrip960</code>	invokes the stripper for backwards compatibility with GNU/960 Release 1.2 and later. On Windows hosts, the invocation command is: <code>gstrip96</code> .
<i>option</i>	is one or more of the options listed in Table 15-1. Use the options only for stripping COFF or ELF files. For b.out files, specify no options.
<i>filename</i>	is one or more files to be stripped.



NOTE. To strip all debugging information for COFF, b.out, or ELF, specify no options. Also, you can interleave the options and filenames in any order.

To strip a file while preserving and re-indexing the relocation entries for incremental linking, use the `r` option. Without `r`, the relocation information inhibits the stripper.

When you use the stripper on an ELF/DWARF file, all sections that lack the `SHF_ALLOC` bit in the section header table are removed from the output file. The one exception to this rule is the `.shstrtab` section, which is not removed.

Table 15-1 **str960/gstrip960 Options**

Option	Effect
a	strips all symbols. This is the default.
b	strips the local symbols and line numbers only, retaining the block information.
C	strips CCINFO only.
h	displays help information and exits.
l	Stips line numbers only.
r	strips files containing relocation information, revising the remaining relocation indexes. External and static symbols remain.
v	displays the stripper version number and the date and time the stripper was created, then continues processing.
v960	displays the stripper version number and the date and time the stripper was created, then stops processing.
x	strips all except static and external symbol information.
z	suppresses writing a time-stamp in the COFF header file.

Partial stripping is supported by the `b`, `C`, `l`, `r` and `x` options. For ELF files, these options are equivalent: Both remove debugging information from the output file. All sections named `.debug*` are removed from the output file, but the file remains relocatable because both the `.strtab` and `.rel*` sections are retained.

Assembly Language Converter (*xlate960*)

16

The *xlate960* program converts assembly language code from 80960 core processors (e.g., i960 Cx, Jx, and Hx processors) to its CORE0 (e.g., 80960Rx) equivalent. *xlate960* performs both instruction translations and addressing-mode translations. Instruction translation occurs when the target architecture does not support a translatable instruction from the source architecture (e.g., *movt*). Addressing mode translation occurs when the target architecture supports a restricted form of an instruction from the source architecture (e.g., *callx*). Table 16-1 lists the instructions converted by *xlate960*.

Table 16-1. Instructions Translated by *xlate960*

<i>addi</i>	<i>lda†</i>	<i>remo</i>
<i>addi<cc></i>	<i>ldib‡</i>	<i>scanbit</i>
<i>balx†</i>	<i>ldist‡</i>	<i>spanbit</i>
<i>bx†</i>	<i>ldl†</i>	<i>st†</i>
<i>calljx‡</i>	<i>ldob‡</i>	<i>stib‡</i>
<i>callx†</i>	<i>ldos†</i>	<i>stis‡</i>
<i>cmpdeci</i>	<i>ldq†</i>	<i>stl†</i>
<i>cmpdeco</i>	<i>ldt‡</i>	<i>stob‡</i>
<i>cmpinci</i>	<i>modac</i>	<i>stos†</i>
<i>cmpinco</i>	<i>modify</i>	<i>stq†</i>
<i>concmpi</i>	<i>movl</i>	<i>stt‡</i>
<i>concmpo</i>	<i>movq</i>	<i>subi</i>
<i>eshro</i>	<i>movt</i>	<i>subi<cc></i>
<i>extract</i>	<i>notor</i>	<i>test<cc></i>
<i>fault<cc></i>	<i>remi</i>	<i>xnor</i>
<i>ld†</i>		

Notes:

† indicates addressing-mode translation

‡ indicates addressing-mode and instruction translation

all other instructions support only instruction translation



NOTE. *Make sure that your code assembles without error for an 80960 core processor before converting with xlate960.*

Invocation

Command-line Invocation

To invoke xlate960 from the UNIX command shell or a Windows DOS prompt, use the syntax:

```
xlate960 [options] infile
```

xlate960	Invokes the converter.
options	Represents any of the options listed in Table 2.
infile	Specifies the name of the assembly file to convert.

Table 16-2. xlate960 Options

Option	Effect
<i>A arch</i>	Specifies output architecture type. <i>arch</i> can be one of: RP, RD, RM, RN, CORE0. Default type is RP.
<i>e</i>	Displays translation compatibility errors to the screen matching those inserted in the output file.
<i>h</i>	Displays help information. <i>xlate960</i> also displays this information if you run the program without specifying the input file and any other options.
<i>l, longcalls</i>	Uses <i>callx</i> instead of <i>call</i> instructions for translation routine library calls.
<i>-o outfile</i>	Specifies output filename. If this option is not given, <i>xlate960</i> replaces an extension of <i>.s</i> or <i>.as</i> with <i>.xlt</i> , or appends <i>.xlt</i> to any other filename.
<i>V</i>	Displays version information and continues processing.
<i>v960</i>	Displays version information and exits.
<i>w</i>	Displays translation compatibility warnings to the screen matching those inserted in the output file.
<i>x</i>	Emits warnings instead of errors for unrecognized instructions.

Invocation Through the Assembler

The translation utility can also be invoked through the assemblers (*gas960e*, *gas960c*, *gas960*, or *asm960*) by using the assembler *-t* option. When invoked in this manner, *xlate960* uses the assembler's architecture specification (*-A* option) as its output target architecture. If errors occur during the translation process, the assembler does not attempt to process the *xlate960* output file. This includes instances where the translator output file requires manual adjustments.

Invocation Examples

The following command converts instructions in `myfile.s` to 80960Rx-compliant instructions, placing the output into the file `myfile.xlt`:

```
xlate960 myfile.s
```

The next example shows how to run `xlate960` from the assembler command line, generating an 80960Rx-compliant object file `myfile.o`:

```
gas960e myfile.s -t -ACORE0
```

Output File Format

The output file produced by `xlate960` is identical to the input file except for the instances where translation occurred. Each instruction that was translated is replaced with a sequence of the following format in the output file:

```
#xlate-begin          original instruction
<translation errors or warnings, marked by xlate-err or xlate-warn>
<translation routine>
#xlate-end
```

For example, given this input file:

```
eshro 10, r5, r5
mov r5, g10
```

`xlate960` produces the following output file:

```
#xlate-begin          eshro 10, r5, r5
#xlate-err           "Fill in register for E0"
shro 10,r5,E0
shlo 22,r6,r5
or r5,E0,r5
#xlate-end
```

```
mov r5, g10
```

User Interaction

Translation Errors

Certain instruction translations may require further action by the user. For example, addressing mode translations often require a scratch register to hold intermediate calculation results. If the translator cannot determine an unused register for this purpose, it emits a translation error into the output file. All translation errors are marked with a comment of the form:

```
#xlate-err          Error message
```

You can display error messages on the screen by using the `-e` switch. If any translation errors occur, the translator displays a message similar to this one at the end of translation:

```
xlate960: Output file 't.xlt' requires further manual
translation.
```

It is the user's responsibility to edit the output file and remove the error conditions before continuing with code migration.

Translation Warnings

Whereas translation errors indicate invalid output, translation warnings indicate more subtle incompatibilities, those that are less likely to affect the compatibility of the output code. An example of this would be a translated code sequence that does not maintain the same faulting behavior as the original code. When the translation utility determines that such a condition exists, it emits a warning into the output file of the format:

```
#xlate-warn        Warning message
```

You can display warning messages on the screen using the `-w` switch. You should also examine translation warnings before continuing with code migration.

Known Limitations

`xlate960` is unable to translate dot-relative expressions in some instances. For example, given the following code fragment:

```
bge .+12
lda 10(r4)[g5*4], g10
mov r5, r6
```

The original intent is that the `lda` instruction is skipped when the AC register condition code indicates a greater than or equal to condition. Because of the line-by-line nature of the translator, however, this code is translated into the following:

```
bge .+12
shlo 2, g5, g10
addo g10, r4, g10
lda 10(g10), g10
mov r5, r6
```

Notice that the `bge` instruction branches into the middle of the translation routine instead of branching around it. These conditions can be avoided by using good programming practices such as creating a label for the target of the `bge` instruction in the above instance.

Linker Command Language



Introduction

The linker command language lets you control the linking process. You can use the linker command language to:

- Define and allocate memory.
- Specify files and libraries to be linked.
- Group sections in memory.
- Place input files' input sections (the smallest relocatable pieces of object files) into output sections (sections in linker output files).
- Resolve global symbols to absolute addresses.
- Calculate checksums.

The linker command language consists of keywords called linker directives. These directives are placed in linker directive files, which are also known as link scripts and configuration files in some environments. Linker directive files typically use the extension `.ld`. You specify linker directive files when you run the linker, either directly or indirectly (via a compiler invocation).

Most of this appendix is devoted to a description of the linker directives. The last section of this appendix, provides some details on linker directive files and sample linker directive files. (The toolset includes linker directive files for various common i960® processor evaluation boards.)

Before using the information in the *Linker Directives Reference* section, you should be familiar with the information in the next section, which discusses other directive file elements, such as expressions and operators.

Expressions and Operators

You can use global symbols, constants, and C-language operators in expressions in linker directive files. Names can contain uppercase or lowercase letters, numbers, dollar signs (\$), and underscores (_). All numbers are long integers. As in C, constants are in decimal format unless preceded by 0 for octal or 0x for hexadecimal.

The value of a symbol corresponds to the value in the symbol table after the link is complete.

Table A-1 shows the operators in order of precedence. Operators on the same level have the same precedence and are processed as encountered.

Table A-1 Order of Precedence for Operators

Precedence	Operators	Associativity
1 (highest)	! - ~ (unary)	left
2	* / %	left
3	+ -	left
4	>> <<	left
5	== != > < <= >=	left
6	&	left
7		left
8	&&	left
9		left
10	?:	right
11 (lowest)	&= *= = += -= /=	right

To define symbols and assign values to them, use assignment statements with the following syntax:

```
symbol [op]= expression;
```

symbol is the symbol name.

op is one of the operators &, *, |, +, -, or /. Not all assignments require an operator.

= is the assignment operator.
expression is a valid expression. For example:
 start + 10



NOTE. *The final semicolon is required for all assignment statements.*

Assignment statements are processed in the order they are input to the linker and are evaluated after allocation. However, for symbols used in the *expression*, the evaluation addresses reflect the symbol addresses in the output object file. Subsequent symbol references access the latest assigned values.

For conditional statements, use the following C syntax:

condition ? *t-expr* : *f-expr*;

condition evaluates to true (1) or false (0). The question mark (?) is required.

t-expr is evaluated when the *condition* evaluates to true.

f-expr is evaluated when the *condition* evaluates to false. The colon (:) separates the *f-expr* from the *t-expr*.

Linker Directives Reference

Use linker directives to:

- Configure your target memory.
- Include library files and other directive files.
- Provide the start-up routine.



NOTE. *In this section, the curly braces ({ and }) are part of the directive syntax and must be entered as shown. Square brackets ([and]) and ellipses (. . .) indicate optional and repeatable elements.*

Table A-2 Linker Directives

Directive	Operation
ADDR	returns a non-relocatable section address.
ALIGN	assigns non-relocatable values to symbols. This operation is different from the ALIGN keyword of the SECTIONS directive. This option is synonymous with NEXT.
CHECKSUM	generates a checksum for the bus confidence test.
DEFINED	returns whether a symbol is defined in the global symbol table.
ENTRY	specifies the first executable instruction address.
[NO]FLOAT	specifies whether to use the extended-arithmetic and floating-point libraries.
FORCE_COMMON_ALLOCATION	forces allocation of space for common symbols, even for non-final (relocatable) linking.
HLL	specifies the high-level support libraries.
INCLUDE	locates and processes the specified directive file.

continued

Table A-2 Linker Directives (continued)

Directive	Operation
INPUT	provides backward compatibility with GNU R2.0.1 (required in that release for naming linker input files inside a directive file). Later versions of the linker allow naming input files without using the INPUT directive.
MEMORY	specifies the available target memory and defines configured and unconfigured memory.
NEXT	synonomous to ALIGN.
OUTPUT	names the output file.
OUTPUT_ARCH	specifies the target 80960 architecture for the current link. Overrides \$G960ARCH and is overridden by the A linker option. Default architecture is KB.
PRE_HLL	Lets the user specify libraries that are processed immediately before the high-level language libraries specified with the HLL () directive.
SEARCH_DIR	extends the library search path (like L <i>dir</i>).
SECTIONS	defines the contents, configuration, and location of the output sections.
SIZEOF	returns a section size.
STARTUP	specifies the first file to be linked.
SYSLIB	specifies the low-level support libraries.
TARGET	uses the search path to find the specified directive file, with the same effect as the T option.

MEMORY: Configuring Memory Regions

Use the MEMORY directive to:

- Specify the target-memory size.
- Designate configured and unconfigured memory regions.
- Prepare memory regions for specific sections.

A

Omitting `MEMORY` configures a single `RWXI` region from `0x0` through `0xffffffff`.

The syntax is:

```
MEMORY {  
    name [(attr)] : origin = expr, length = expr  
    [. . .]  
}
```

<code>name</code>	is a symbol for an address range.
<code>(attr)</code>	is one or more of the attributes listed in Table A-3. The parentheses are required when you specify <code>attr</code> .
<code>origin</code>	is a keyword assigning the region starting address. You can abbreviate <code>origin</code> to <code>org</code> or <code>o</code> . Use either a space or a comma to separate <code>origin</code> and <code>length</code> .
<code>length</code>	is a keyword assigning the region size, in bytes. You can abbreviate <code>length</code> to <code>len</code> or <code>l</code> .
<code>expr</code>	is a decimal, octal, or hexadecimal expression in C syntax.



NOTE. *When you prepare memory regions for sections or groups with `(attr)`, the `MEMORY` attribute list must exactly match the corresponding `SECTIONS` attribute list. Omitting the attribute list gives the memory region the `RWXI` attributes.*

Table A-3 Memory and Section Attributes

Attribute	Characteristic
R	indicates a readable region.
W	indicates a writable region.
X	indicates an executable region.
I, L	indicate a region that can be initialized.



NOTE. *Specifying more memory than is actually contained in your system causes a fatal error.*

For more information on assigning sections and groups to memory regions, see the *SECTIONS: Defining Output Sections* section.

Examples

1. The following example prevents any code from being located in the first 0x10000 words of memory:

```
MEMORY {
    valid : org = 0x10000, len = 0xffff0000
}
```

2. In the following example the largest configured region is 0x10000 bytes:

```
MEMORY {
    mem1:  o = 0x00000  l = 0x02000
    mem2:  o = 0x40000  l = 0x05000
    mem3:  o = 0x20000  l = 0x10000
}
```

Default Linker Allocation

The term allocation refers to the process the linker uses to locate input sections into output sections and then to assign the output sections to actual memory addresses.

Omitting a `SECTIONS` directive puts all input sections with the same name in an output section of that name, in the order shown below. For example, linking several object files, each containing the `.text`, `.data`, `.mysect`, and `.bss` input sections, creates the combined `.text`, `.data`, `.mysect`, and `.bss` output sections. The linker allocates the output sections in the following order when a `SECTIONS` directive is not supplied:

- `.text` at the lowest available address
- other text-type sections after `.text`
- the `.data` section after all the text-type sections
- other data-type sections after `.data`
- other types of sections after all the text-type and data-type sections
- `.bss` after all other sections

If you provide a `SECTIONS` directive, the linker first locates any sections you specify, then puts unassigned sections into suitably configured memory on a first-fit basis, keeping groups intact.

You should carefully study the link map after you link your executable. If you do not like the arrangement, use the techniques discussed below to give the linker explicit directions on how to allocate your application. You can use the `v` linker option to display the order in which it allocates output sections and the addresses it chooses for the sections.

SECTIONS: Defining Output Sections

Use the `SECTIONS` directive to:

- combine the input sections into output sections
- locate the output sections in memory
- create and initialize symbols
- initialize unassigned memory
- locate the entry point

The syntax is:

```
SECTIONS {
  [GROUP [addr-spec]: {}] # opens an optional GROUP block
    o-section [addr-spec] [ns-type]: {
      [statements]
    } [=fill] [>mem-attr]
    [...]
  [] [mem-attr] # closes the optional GROUP block
  [...]
}
```

GROUP	specifies sections to be treated as one unit. See Table A-4.
<i>addr-spec</i>	is a starting address. For more information on this subject see the description for the B and T section start address options in Chapter 7.
<i>o-section</i>	defines an output section. The colon is required.
<i>ns-type</i>	is the DSECT, NOLOAD, or COPY nonstandard type (these may have optional parentheses).
<i>statements</i>	describe the section contents, including filenames, input-section names, keywords (listed in Table A-4), assignments, and other expressions (described in the <i>Linker Command Language</i> section). The braces are required, even with no statements. An empty output-section definition contains all otherwise-unallocated input sections with the same name (<i>o-section</i>) as the output section.
<i>fill</i>	is a two-byte initialization value for spaces between input sections in the output section.

A

mem-attr specifies a memory region either by name or by attribute, as described in the *MEMORY: Configuring Memory Regions* section. The > is required. For a list of the attributes, see Table A-4.



NOTES. Specify a starting address or a memory region (or neither), but not both. Conflicting group and section specifications cause syntax errors.

In a *SECTIONS* directive, you can use the keywords summarized in Table A-4.

Table A-4 SECTION Keywords

Keyword	Operation
ALIGN	returns the next address that fits the specified boundary.
BYTE	puts a byte value at the current address.
COMMON	locates uninitialized data.
COPY	copies output-section data to the output file without relocating the section or allocating any memory.
CREATE_OBJECT_SYMBOLS	creates a symbol for each input file.
DSECT	creates an empty section, allocating no memory (but processing contained symbols).
ENTRY	locates the entry point.
FILL	specifies the value used to fill gaps in an output section.
GROUP	specifies sections to be treated as a unit.
LONG	puts a 4-byte (word) value at the current address.
NOLOAD	allocates memory and locates the output section, but copies no data to the output file.
SHORT	puts a two-byte (half-word) value at the current address.

The sample below shows a `SECTIONS` directive where all `.text`, `.data`, and `.bss` sections are allocated to DRAM.

```
SECTIONS
{
    .text :
    {
        } >dram

    .data :
    {
        } >dram

    .bss :
    {
        } >dram
}
```

Combining Input Sections Into Output Sections

By default, the linker combines sections as follows:

- Combine all input sections of each name together into one output section with the input-section name. In each output section, sequence the input sections in the order encountered in the input files.

You can build an output section from specific input sections. List the input filenames and sections in an output-section definition as:

```
[filename]( [i-sections])
```

filename is an input filename.

i-sections is one or more input-section names, separated with commas or spaces.

You can use the wildcard character (*) for the filename. For example:

*(.text) specifies all input file `.text` sections.

Common Sections (COMMON)

The term common section refers to global uninitialized variable space. The following are examples of common variables:

```
int x,y[20];
```

With respect to impact on external linkage, this term is used exactly as it is in the FORTRAN programming language. COMMON symbols are overridden by a larger definition of the same name, or by a definition that is initialized. For example:

```
int x;                defined in x.o
double x;            defined in y.o
char x = 'a';        defined in z.o
```

would each override each other resulting in a global variable named `x` that is an initialized character variable. To designate an output section for common sections of input files, use the `COMMON` keyword anywhere in an output-section definition. To include the common symbols from specific input sections, use `COMMON` after the input-section statement. For example:

```
foo.o(COMMON) /* COMMON section from foo.o */
```

Similarly, the wildcard can be used:

```
*(COMMON) /* Place ALL COMMON sections here. */
```

`COMMON` tells the linker where to locate variables not assigned to another section. It can refer to all such variables or just to the unassigned variables from specific input files.

You can list the input files in the output symbol table. To create a symbol for each input file in an output section, specify `CREATE_OBJECT_SYMBOLS` at the beginning of the output-section definition, after the opening brace:

```
o-section [addr-spec]: { CREATE_OBJECT_SYMBOLS
    { [statements] }
}
```

Note that `*(section_name)` and `[section_names]` mean that if any input sections named `section_name` exist, and are not already placed in an output section, place them in this output section. For example, the following code produces an error if `bar.o` is the only input object file.

```
SECTIONS {
foo {bar.o}
junk : { *(COMMON) }
}
```

However, the following example does not produce an error because as it is evaluating `*(COMMON)`, `bar.o(COMMON)` is available.

```
SECTIONS {
junk : { *(COMMON) }
foo : { bar.o }
}
```

Files and sections are bound top to bottom, and left to right in linker directive files.

Splitting COFF Output Sections

The common object file format (COFF) allows only 65535 line-number entries and 65535 relocation entries for any section within an object file. The linker creates additional output sections whenever adding another input section to an existing output section overflows the line number entry table or the relocation entry table.

To split a section, the linker:

- creates a new section with the original section name, starting with the first input section that does not fit completely in the original section.
- assigns a new name to the original output section by appending a number to the original section name, truncated as necessary to limit the new section name to eight characters.

For example:

- Splitting the `.mybgtxt` section once assigns the name `.mybgtx0` to the original output section and creates a new section called `.mybgtxt` for the remaining input-section contents. More than 10 splits create section names such as `.mybgt10` and `.mybgt11`.

- A `.text` section of 196700 bytes splits into four sections named `.text0` (containing the first input sections), `.text1`, `.text2`, and `.text` (containing the last input sections).

The linker notifies you of each section split with a warning message unless you suppress all warnings by using the `w` linker option. You receive no other indication of section splits unless you produce a memory map by using the `m` linker option.



NOTE. *The linker treats the new sections created under the same linker directives that applied to the original section. Do not put the new names into the directive file.*

Examples

1. The following creates an output section named `.ab_txt`, containing the input `.text` sections from all the available input files, all the input sections from the `abn.o` file, the `.atxt` and `.atxtq` sections from `afile.o`, and `.btxt` from `bfile.o`, located as described in the *MEMORY: Configuring Memory Regions* section.

```
SECTIONS {
    .ab_txt: {
        *(.text)
        abn.o
        afile.o(.atxt .atxtq)
        bfile.o(.btxt)
    }
}
```

2. The following puts all of the common symbols from all the input files into the `mycom` output section:

```
SECTIONS {
    mycom: {[COMMON]}
}
```

3. To place `COMMON` sections from a specific file into a particular section, use this form:

```
SECTIONS {
    xxx : { filename.o(COMMON) }
}
```

4. The following puts a symbol in the output symbol table for each file that contributes a `.txt` input section to the `.utxt` output section:

```
SECTIONS {
    .utxt: {
        CREATE_OBJECT_SYMBOLS
        *(.txt)
    }
}
```

If `COMMON` is not in the linker directive file in a final link, uninitialized variables are placed by default into the `.bss` section.

Allocating the Output Sections (>*region*, `ALIGN`, `GROUP`)

You can allocate an output section by:

- explicit addresses
- alignment
- memory region name
- memory attribute

When allocating the output sections, the linker maintains the default or specified alignment. The linker processes the output sections in the following order:

1. output sections with explicit starting addresses.
2. output sections with memory-region names or attributes, placing each section into the first appropriately configured, unallocated region of sufficient size.
3. output sections with no location specifications, placing each section into the first appropriate unallocated region of sufficient size.

For information on configuring memory regions, see the *MEMORY: Configuring Memory Regions* section.

By default, the linker uses the largest alignment of any input section in the output-section definition for the entire output section. You can specify a larger output-section alignment, which then overrides the input section's alignment specifications.

Specify the section alignment or address (not both) after the name of an output-section specification, before the colon and the opening brace:

```
o-section [ ALIGN(align-expr)
             addr-expr ] : { [statements] }
```

align-expr evaluates to a power of 2.

addr-expr evaluates to an address in configured memory.

Instead of allocating the output sections by an address, you can assign an output section to a specific memory region by name or by attribute. For a region name, use the > operator and the name after the output-section definition:

```
o-section: { [statements] } [>region]
```

region is a memory-region name or attribute list, as defined in a MEMORY directive. The > is required. For more information on matching sections and memory regions by attribute, see the *MEMORY: Configuring Memory Regions* section.



NOTE. *You cannot specify both an address and a memory region.*

Treating Output Sections as a Unit (GROUP)

You can specify a set of output sections to be treated as a unit, including:

- contiguous memory locations, with the order of the sections preserved
- homogeneous attributes, kept together in a single memory region

Put the output sections in a `GROUP` block in the `SECTIONS` directive. You can define multiple groups:

```
GROUP [addr-spec]: {
    section-defs
} [>mem-spec]
```

`addr-spec` specifies an alignment or address. For more information on this subject see the description for the `B` and `T` section start address options in Chapter 7.

`section-defs` is one or more output-section definitions.

`mem-spec` specifies a memory region by name or attributes. The `>` is required.



NOTES. *Specify the address or memory region for a group the same as for a section. Specify no such locations for the individual sections within a group.*

An output section or group that does not fit at the specified address causes an error.

Although the `GROUP()` directive should keep the sections in the order specified, empty sections that contain no external symbols and are not user-defined (e.g., `.data`, `.text`, `.bss`) do not always appear in the specified order. This behavior can be avoided by declaring an external symbol in the empty section.

Although the linker does not ensure that the sections are an even number of bytes in length, you need not align the individual input sections within an aligned output section. The assembler and compiler create sections that are multiples of four bytes in length.

Examples

1. The following aligns the `outsec` output section on an address that is a multiple of `0x20000` (address `0x0`, `0x20000`, `0x40000`, etc.)

```
SECTIONS { outsec ALIGN(0x20000): { } }
```

2. The following assigns `.text` the starting address `0x04000000`, `.data` the first available address in `mem1`, and `.bss` the first memory location big enough hold it:

```
MEMORY {
    mem1: o = 0x10000000, l = 0x20000
    mem2: o = 0x40000000, l = 0x40000
}*
SECTIONS {
    .text 0x04000000: {}
    .data: {} > mem1
    .bss: {}
}
```

3. The following keeps the `.text`, `.data`, and `.bss` sections together in the `membase` memory region. Note that all of the sections are aligned per the input section requirements and that `.data` immediately follows `.text`, and `.bss` immediately follows `.data`.

```
SECTIONS {
    GROUP: {
        .text : { }
        .data : { }
        .bss  : { }
    } >membase
}
```

Creating Gaps and Defining Symbols in Output Sections (ALIGN, BYTE, FILL, LONG, SHORT, dot)

A section gap contains no information. To create a gap:

1. Change the location counter, represented by the dot (`.`) symbol, to insert a gap of any length filled with a repeated two-byte value.
2. Use an initialization keyword to insert a byte, half-word, or word value.

Assign a new value to the location counter with either:

```
. [operator]= size-expr;
. = ALIGN(align-expr);
```

operator is an +, -, *, or / operator.

size-expr evaluates to an offset address, relative to the beginning of the output section, or to a value to be used by the operator.

align-expr evaluates to a power of 2.

ALIGN returns the next address, within the output section, that is divisible by the *align-expr*.

To specify a repeating two-byte initialization value for the gaps in a section, assign the fill value after the closing brace of the output-section definition (or use FILL () within a section definition):

```
o-section [addr-spec]: {
    [statements]
} [=fill] [mem-spec]
```

The phrase =*fill* designates the fill value for the *o-section* output section.

For one-, two-, or four-byte gaps, use the initialization keywords:

```
BYTE(expr)
SHORT(expr)
LONG(expr)
```

expr is a byte, half-word, or word value, respectively. For any value that is longer than the keyword requires, the linker uses the least-significant byte, half-word, or word.

The location counter within an output-section definition is an offset relative to the base address of the output section. Any address or alignment you specify is relative to the beginning of the output section. To find or specify the location counter absolutely, specify the base address by explicitly locating the output section, as described in the *Locating the Output Sections* (>region, ALIGN, GROUP) section.

Examples

1. The following puts two gaps in the `outsec` output section:
 - A 0x1000 byte gap is left at the beginning of the `outsec` output section. The `f1.o(.text)` input section begins after the gap.
 - The `f2.o(.text)` input section begins at 0x100 bytes after the end of `f1.o(.text)`.
 - The `f3.o(.text)` input section begins on the next word boundary with respect to the beginning of `outsec`.
 - The gaps are filled with the two-byte value 0x0020.

```
SECTIONS {
    outsec: {
        .+= 0x1000;
        f1.o(.text)
        .+= 0x100;
        f2.o(.text)
        . = ALIGN (4);
        f3.o(.text)
    }=0x0020
}
```

2. The following quad-word-aligns `outsec`:

```
SECTIONS {
    outsec ALIGN(16): {
        .+= 0x1000;
        f1.o(.text)
        .+= 0x100;
        f2.o(.text)
        f3.o(.text)
    }
}
```

3. In this example:

- The `s2_start` symbol points to the beginning of `infile2(ss2)` (section `ss2` from input file `infile2`)
- The `s2_end` symbol points to the last byte of `infile2(ss2)`.

```
SECTIONS {
    outsc1: {
        infile1(ss1)
        s2_start = . ;
        infile2(ss2)
        s2_end = .-1;
    }
}
```

4. In this example, the symbol `mark` points to the first full word beyond the end of the `.data` section of `file1.o`. Four bytes are reserved by the `. += 4;` statement for a run-time initialization of `mysymbol`, representing a long integer.

```
SECTIONS {
    outsc1: {
        file1.o (.data)
        . = ALIGN(4);
        mysymbol = .;
        . += 4;
        file2.o (.data)
    }
}
```

5. This example does the following:

- aligns the `.data` output section on the next 0x2000-byte boundary after the `.text` output section
- defines the `als` symbol to point to the next 0x8000-byte boundary after the `.data` input sections

```
SECTIONS {
    .text { }
    .data ALIGN(0x2000): {
        *(.data)
        als = ALIGN(0x8000);
    }
}
```

Defining Non-standard Sections (DSECT, COPY, NOLOAD)

You can create output sections that:

- overlay other sections
- contain complete and exact copies of the input sections
- contain no data

Specify the non-standard section keywords (with or without parentheses) immediately before the colon and opening brace of an output-section definition:

```
o-section [addr-spec]  $\left[ \begin{array}{l} \text{DSECT} \\ \text{COPY} \\ \text{NOLOAD} \end{array} \right] : \{ [\textit{statements}] \}$ 
```

With `DSECT`, you can locate the symbols in an output section without writing any object code to the output file or allocating any memory. The global symbols defined in a `DSECT` section are relocated normally, are resolved as needed from the libraries, and are available to other sections. For example, use `DSECT` to create an overlay section that, during execution, can re-use a memory region no longer needed by a prior section. Overlay section data can be read in from peripheral storage during execution.

To exclude the input-section contents from the output section, use `NOLOAD`. A `NOLOAD` output section has no data in the object file, but occupies memory and appears in the memory map. For example, use `NOLOAD` to reserve a memory region for a section that is linked and located separately.

To copy the input-section contents and all associated information to the output file, use `COPY`. A `COPY` output section is not located and occupies no memory.

Example

The following allocates one of each nonstandard output-section type:

```
SECTIONS {
    name1 0x200000 DSECT: { file1.o }
    name2 0x400000 COPY: { file2.o }
    name3 0x600000 NOLOAD: { file3.o }
}
```

COFF Binary Representations

The COFF binary representations of the NOLOAD, DSECT and COPY sections are detailed in the table below:

Table A-5 COFF Binary Representation of NOLOAD, DSECT, COPY Sections

	ALLOC	RELOCATED	scnptr	size	flags	LOADED
NOLOAD	N	NA	0	S	NOLOAD	N
DSECT	N	NA	0	0	DSECT	N
COPY	N	N	S	S	COPY	Y
ORDINARY	Y	Y	S	S	0	Y

The `ALLOC` column indicates whether or not the linker allocates memory for it.

`RELOCATED` indicates whether the section is relocated or not. Since the `DSECT` and `NOLOAD` sections do not have section contents, it is not applicable to them. Note that the `COPY` sections are not relocated, but copied verbatim.

If `scnptr` is 0 in COFF, there are no section contents for it, even if there is size of the section. `S` indicates that the `scnptr` corresponds to the file seek address of the section contents. See Appendix C for information on the the COFF OMF.

If `size` is 0, the output section for that element is filled with zeros. `S` indicates that the section size is retained for your information.

A

Some of the sections set some special flags into the section's flagword. The table above indicates which flags are set.

A loader can load anything from the OMF file but, according to the semantics defined here, the loader loads only those marked with a Y in the table above.

ELF Binary Representations

The ELF binary representations of the `NOLOAD`, `DSECT` and `COPY` sections are detailed in the table below:

Table A-6 ELF Binary Representation of `NOLOAD`, `DSECT`, `COPY` Sections

	<code>SHF_ALLOC</code>	<code>RELOCATED</code>	<code>SHT_PROGBITS</code>	<code>SIZE</code>	<code>HAS_PROGRAM_HDR</code>
<code>NOLOAD</code>	N	NA	N	S	N
<code>DSECT</code>	N	NA	N	0	N
<code>COPY</code>	Y	N	Y	S	Y
<code>ORDINARY</code>	Y	Y	Y	S	Y

`SHF_ALLOC` indicates whether or not the section is allocated memory by the linker. It also indicates that the output flag word contains this output flag.

`RELOCATED` indicates whether the section is relocated or not. Since the `DSECT` and `NOLOAD` sections do not have section contents, it is not applicable to them. Note that the `COPY` sections are not relocated, but copied verbatim.

Each section contains `SHT_PROGBITS` or `SHT_NOBITS`. Y indicates that the section is a `SHT_PROGBITS`, while N indicates it is `SHT_NOBITS`.

If `size` is 0, the output section has zeroed that element. S indicates that the section size is retained for your information.

If the column for `HAS_PROGRAM_HDR` contains a Y, the section should be loaded by a memory loader utility.

FORCE_COMMON_ALLOCATION: Allocating Space for Common Symbols

To assign common-symbol space in the output data, use `FORCE_COMMON_ALLOCATION`. This directive has the same effect as the `d` option. You can use `FORCE_COMMON_ALLOCATION` when generating either relocatable or non-relocatable linked files. However, this feature is most useful when using relocatable links.

DEFINED: Finding Symbols

To determine whether a global symbol is defined, use `DEFINED`:

```
DEFINED(symbol)
```

symbol is the symbol name.

Finding the symbol in the global symbol table returns 1.

In the following example, the value of `begin` is preserved if `begin` already exists in the global symbol table; otherwise, `begin` is set to the location counter (`.`):

```
begin = DEFINED(begin) ? begin : . ;
```

ADDR, ALIGN, NEXT: Finding Addresses

To find the absolute beginning address of a section, use `ADDR`:

```
ADDR(section)
```

section is the name of a located section.

For an address aligned after the current location counter, use `ALIGN` or `NEXT`:

```
ALIGN(expr)
```

```
NEXT(expr)
```

expr is an alignment factor.

For memory with no unconfigured regions, `ALIGN` and `NEXT` are equivalent. `ALIGN` returns the next address in configured memory that fits the specified boundary. `NEXT` returns the next unallocated address that fits the boundary.

Examples

1. The following locates the `osec1` output section in the `mem1` memory region and assigns the `osec1` beginning address to the `begin_1` symbol:

```
SECTIONS {
    osec1 : { *(.osec1) } >mem1
    begin_1 = ADDR(osec1);
}
```

2. The following assigns the first word-aligned address after the location counter to the `mark1` symbol. If `osec1` completely fills `mem1`, the `mark1` value is `0x02000`, in unconfigured memory:

```
MEMORY {
    mem1: o = 0x00000 l = 0x02000
    mem3: o = 0x40000 l = 0x05000
}
SECTIONS {
    osec1: { } >mem1
    mark1 = NEXT(4);
}
```

3. The following assigns the first word-aligned address after the location counter to the `mark1` symbol. If `osec1` completely fills `mem1`, the `mark1` value is `0x40000`, in the next configured memory region:

```
MEMORY {
    mem1: o = 0x00000 l = 0x02000
    mem3: o = 0x40000 l = 0x05000
}
SECTIONS {
    osec1: { } >mem1
    mark1 = ALIGN(4);
}
```

SIZEOF: Finding Section Sizes

To find the size, in bytes, of a section, use `SIZEOF`:

```
SIZEOF(section)
```

section is the name of a located section.

The following example locates and sizes the `.out1` output section. The `outsz1` and `outsz2` symbols acquire identical values.

```
SECTIONS {  
  .out1: {  
    st_o1 = . ;  
    *(.out)  
    end_o1 = . ;  
  }  
  outsz1 = end_o1 - st_o1;  
  outsz2 = SIZEOF(.out1);  
}
```

STARTUP: Specifying a Startup File

The syntax for the `STARTUP` directive is:

```
STARTUP(filename)
```

filename specifies the file to be linked first.

Specifying a file with `STARTUP` links the file first. This is similar to `SYSLIB`, except that with `SYSLIB` the file is linked after all other object files and libraries. See page A-31 for more information on `SYSLIB`.

To find the file specified with `STARTUP`, the linker follows the search algorithm described in the *Search Paths* section.

The `C` (`startup`) linker option overrides the `STARTUP` directive, as described in the *Option Reference* section.

You can also use an asterisk to instruct the linker to search multiple libraries. For example, if you specify:

```
STARTUP yourlib*
```

the linker searches for `yourlib`, `yourlib_b`, `yourlib_p`, and `yourlib_e`.

ENTRY: Defining the Entry Point

The linker selects an entry point with the following order of precedence:

1. the symbol you specify with the `e` option
2. the symbol you specify with the `ENTRY` directive
3. the `start` symbol, if defined
4. the `_main` symbol, if defined
5. the first address in `.text`
6. the address `0`

You can assign the entry point with `e`, `ENTRY`, or an assignment to `start` or `_main`. For example:

```
start = _my_start_function;
```

You can use `ENTRY` anywhere in your directive file, including inside an output-section definition.

PRE_HLL(): Specifying Libraries to be Processed Before the High-level Libraries

The syntax for the new directive is:

```
PRE_HLL(libraries)
```

libraries

is an abbreviation for one or more high-level support libraries to be linked prior to those specified with an `HLL()` directive. The common library abbreviations are described on page 7-21.

The linker directive `PRE_HLL()` allows the user to specify libraries that are processed immediately before the high-level language libraries specified with the `HLL()` directive. The linker now loads the object files and libraries in the following order:

1. The filename specified with `STARTUP`.
2. All the object files and libraries listed individually in the invocation, in the order listed.
3. All the object files and libraries listed individually in the directive files, in the order listed.
4. All the libraries specified with `PRE_HLL`.
5. All the libraries specified with `HLL` or default libraries in response to `HLL()`.
6. All the libraries specified with `SYSLIB`.

HLL: Specifying High-level Libraries

The syntax for the directive is:

```
HLL([libraries])
```

libraries is one or more high-level support library filenames. The parentheses are required.

If you do not specify libraries, the default HLL libraries are used.

Specify multiple libraries by entering more than one `HLL` directive or by entering multiple filenames separated by spaces or commas. To use the default libraries, enter `HLL()`.

The libraries and search path found by `HLL` depend on the linker invocation and the output format. A `g1d960` invocation for COFF treats the `HLL` arguments the same as the `l` linker option arguments, with the search algorithm described in the *Search Paths* section. The default COFF library abbreviations for `HLL()` are:

- `c` for the KB and SB architectures
- `c` and `fp` for the KA, SA, C-series, and J-series architectures

Invoking the linker as `lnk960` has the following effect on `HLL`:

- Any arguments you specify must be the full library filenames.
- The linker uses the search path for `lnk960` invocations, as described in the *Search Paths* section.
- When you specify `HLL()` with no arguments, the linker includes a high-level C library abbreviated as `libc` and the 64-bit integer support library abbreviated as `libu`. Also, specifying `FLOAT` includes:
 - the `libm` high-level math library abbreviation, for all architectures.
 - The `libh` floating-point library abbreviation, for the i960® KA, SA, Cx, Jx, Hx, and Rx architectures.

Without `FLOAT`, or with `NOFLOAT`, the linker uses `libmstub.a` instead of any `libm` library and includes no floating-point library.

To form the library filenames, the linker appends the following to the `libc`, `libm`, and `libh` abbreviations:

1. the two-letter architecture abbreviation
2. `_p`, for position independence
3. `.a`, the standard library-filename extension

For example:

- The C library for a non-position-independent KB program is `libckb.a`.
- The math library for position-independent KB program, with `FLOAT` specified, is `libmkb_p.a`.
- The floating-point library for position-independent KA output, with `FLOAT` specified, is `libhka_p.a`.

The linker loads the object files and libraries in the following order:

1. the gap specified with `STARTUP`.
2. all the object files and libraries listed individually in the invocation, in the order listed.
3. all the object files and libraries listed individually in the directive files, in the order listed.

4. all the libraries specified with HLL or default libraries in response to HLL().
5. all the libraries specified with SYSLIB.

SYSLIB: Specifying Low-level Libraries

The syntax for the directive is:

```
SYSLIB(libraries)
```

libraries is one or more libraries to be linked last.

Specify multiple libraries by entering more than one SYSLIB directive or by entering multiple names separated by commas or spaces. You must put parentheses around the filenames. The linker follows the search algorithm described in the *Search Paths* section. You can also use an asterisk to instruct the linker to search multiple libraries. For example, if you specify:

```
SYSLIB yourlib*
```

the linker searches for `yourlib`, `yourlib_b`, `yourlib_p`, and `yourlib_e`.

[NO]FLOAT: Supporting Floating-point Operations

The syntax for the directive is:

```
FLOAT | NOFLOAT
```

FLOAT specifies support for floating-point operations. The linker loads special emulation libraries for the i960® KA, SA, Cx, Jx, Hx, and RP processors, which have no on-chip floating-point support. For all processors, the linker also loads an extended math support library. For information on the floating-point and math library names, see the *Directives Reference* section.

`NOFLOAT` indicates no need for floating-point operations. The linker links the `libmstub.a` rudimentary math library and no emulation libraries. `NOFLOAT` is the default.

SEARCH_DIR: Extending the Search Path

To extend the linker search path, use `SEARCH_DIR`:

```
SEARCH_DIR(path)
```

path is a directory to be searched for libraries.

For a complete description of the library search path, see the *Search Paths* section.

The `L` option has the same effect as the `SEARCH_DIR` directive.

INCLUDE: Including Additional Directive Files

The syntax for the directive is:

```
INCLUDE(config-file)
```

config-file is the name of the linker directive file.

You can specify the file to be included with a full pathname or with a filename. The search algorithm differs according to the way you enter the file specification. When you enter a full or relative pathname, the linker searches only the specified directory for the file. When you enter only a filename, the linker searches for the file as follows:

- The `INCLUDE` directive is within a target file, that is, a file specified with the `TARGET` directive or the `T` (target) linker option. The linker searches for the file according to the algorithm described in the *Search Paths* section.
- The `INCLUDE` directive is in any other directive file. The linker searches only the current directory for the specified file.

TARGET: Using the Search Path for Directive Files

The syntax for the directive is:

```
TARGET(filename)
```

filename is the directive filename for your target system.

The linker follows the search algorithm described in the *Search Paths* section to find a directive file.

The TARGET directive has the same effect as the T linker option.

CHECKSUM: Preparing for the Bus Confidence Test

The syntax for the directive is:

```
last-check-word=CHECKSUM(expr, . . .)
```

expr identify the checksum component values.

Use this directive to obtain the value for the final checksum word to complete the bus confidence test, as explained in your processor user's manual. Typically, you want to load the first few check words with bit patterns that expose possible bus failure symptoms. CHECKSUM lets you load the final check word with a value that results in zero when the processor performs an add-with-carry on all words in the initialization boot record (IBR).

The CHECKSUM directive takes a variable number of arguments, depending upon the target processor's requirements. You define the values in the directive file for your program.

For example, on an i960 CA processor, suppose the IBR contains the following word directives:

```
_init_boot_record:
    .word BYTE_0 (BUS_CONFIG)
    .word BYTE_1 (BUS_CONFIG)
    .word BYTE_2 (BUS_CONFIG)
    .word BYTE_3 (BUS_CONFIG)
    .word _start
    .word _rom_prcb
    .word cwd1
    .word cwd2
    .word cwd3
    .word cwd4
    .word cwd5
    .word cwd6
```

In the directive file, you define `cwd1` through `cwd5`, then use the `CHECKSUM` directive to determine the value of `cwd6`, as follows:

```
cwd1 = 0xa5a5a5a5;
cwd2 = 0xffffffff;
cwd3 = 0x55555555;
cwd4 = 0xaaaaaaaa;
cwd5 = 0x5a5a5a5a;
cwd6 = checksum(_start, _rom_prcb, cwd1, cwd2, cwd3, cwd4, cwd5);
```

For the `CHECKSUM` definition for a particular target processor, see the chapter on initialization in the appropriate processor user's manual.

OUTPUT: Naming the Output File

To specify an output filename other than the default, use `OUTPUT`:

```
OUTPUT(filename)
```

filename is the output filename.

The default output filenames are:

a.out	for COFF format output
b.out	for b.out format output
e.out	for ELF format output

The `OUTPUT` directive has the same effect as the `o` linker option.

Linker Directive Files

To avoid reentering a long or often used invocations, keep options and filename information in a text file containing linker directives. Linker directive files typically use the extension `.ld`. For example, the following specifies `file1.o`, `file2.o`, and `file3.o` in the linker invocation:

```
lnk960 -f 0xffff file1.o file2.o file3.o
```

The following directive file, named `lnkcmd.ld`, specifies the same filenames and `f` option:

```
-f 0xffff  
file1.o  
file2.o  
file3.o
```

Using `lnkcmd.ld` shortens the linker invocation:

```
lnk960 lnkcmd.txt
```

You can put options, object filenames, and library filenames in directive files. To nest directive files, use `INCLUDE`. Precede options in directive files with a hyphen (`-`), not a slash (`/`), regardless of your host system. The linker processes filenames and `INCLUDE` directives in the order encountered.

Linker directive files support comments that are delimited by `/*` and `*/` (just as in C).



NOTE. *You cannot use a hyphen (-) as the first character of a filename.*

In the following example, the `ifile.txt` file contains:

```
file3.o
file4.o
```

Linking occurs in the order `file1.o`, `file3.o`, `file4.o`, `file2.o` when you enter:

```
lnk960 file1.o ifile.txt file2.o
```

Example

The following is an example linker directive file suitable for use with a fictitious target board. See the other linker directive files shipped with the tools located in `$I960BASE/lib/*.ld` and `$G960BASE/lib/*.ld` for more examples.

```
/* You can include invocation options at the beginning of
   the linker directive file, for shorter, more
   consistent linker invocations. */
-ACA /* Produce code for an Intel 960 CA processor. */
-m /* Produce a map file. */

-N map.out /* Write the map file to the file map.out */
-v /* Produce verbose output. */

/* You can specify input modules at the beginning of the
   directive file, to be processed as if on the
   invocation line. You can also include in the
   invocation a separate text file containing only input
   filenames, one per line, to be processed as if on the
   invocation line or at the beginning of the directive
   file. */
file1.o
file2.o
file3.o
```

```
MEMORY {
    ibr:    o=0xffffffff00,l=0x00ff /* The Intel 80960 CA
        Initial Boot Record. */
    rom:    o=0xffff8000,l=0x7800 /* Assume a bank of ROM
        exists at this address. */
    ram:    o=0xe0ff9000,l=0x6000 /* Assume some RAM exists
        at this address. */
}

SECTIONS {
/* We allocate the ibr to ibr memory. Assume the code for
the ibr is in the input section .text in the file named
ca_ibr.o. */
    ibr: {
        ca_ibr.o(.text)
    } > ibr

/* Assume we want the executable code and constants found
in the .text input sections allocated to the rom memory */
    .text : { *(.text) } > rom

/* We allocate the .data sections to ram. */
    .data : {
        *(.data)
    } > ram

/* We allocate the .bss section to ram (following the end
of .data. We also place all common variables here.
Lastly, note how we save off the addresses of the start of
bss and the end of bss, for possible later use at runtime.
*/
```

A

```
.bss : {_start_bss = . ;
      *(.bss)
      [COMMON]
      _end_bss = . ;
} > ram
}

SYSLIB(mylibca.a) /* We include a system library in the
                  linkage. */
```

Finding Information in Object Files

B

Using the Common Object File Library: COFL

To use a function from the `libld960.a` common object file library (COFL), call the function from your C source text and link the assembled object file with the COFL. Put the following lines in your C source text before the first COFL function call:

```
#include <stdio.h>
#include "ldfcn.h"
```

Add the directory containing `ldfcn.h` to your host-system compiler search path. For more information on your host-system compiler, see your host-system documentation. For more information on the `.h` header files and directories, see the *i960® Processor Library Supplement*.

The COFL includes the functions listed in Table B-1.

Table B-1 Common Object File Library (COFL) Functions

Function	Definition
<code>ldaclose</code>	Closes the object file or archive.
<code>ldahread</code>	Reads an archive member header.
<code>ldaopen</code>	Opens the object file or archive for reading.
<code>ldclose</code>	Closes the object file or archive member.
<code>ldfhread</code>	Reads the file header.
<code>ldgetname</code>	Retrieves a name from the object file symbol table.
<code>ldlinit</code>	Prepares the object file for reading line number entries via <code>ldlitem</code> .

continued 

Table B-1 Common Object File Library (COFL) Functions (continued)

Function	Definition
ldlitem	Reads the line number entry from the object file after ldinit.
ldlread	Reads the line number entry from the object file.
ldlseek	Seeks to the line number entries.
ldlnseek	Seeks to the line number entries, given the section name.
ldnrseek	Seeks to the relocation entries, given the section name.
ldnshread	Reads the section header, given the section name.
ldnsseek	Seeks to the section, given the section name.
ldohseek	Seeks to the optional file header.
ldopen	Opens the object file or archive member for reading.
ldrseek	Seeks to the relocation entries.
ldshread	Reads the section header, given the section number.
ldsseek	Seeks to the section.
ldtbindex	Returns the long index of the symbol table entry at the current position.
ldtbread	Reads a specific symbol table entry.
ldtbseek	Seeks to the symbol table

Extracting File Header Information

To extract COFF file-header information, use one of the macros listed in Table B-2. Each header information macro takes as an argument an `ldfile` structure returned by a call to `ldopen`.

Table B-2 Common Object File Interface Macros

Macro	Definition
TYPE	Returns the file type number. For archive files, TYPE returns ARTYPE, as defined in ldfcn.h
IOPTR	Returns the file pointer opened by ldopen and used by the C library I/O functions.
OFFSET	Returns the object file beginning address. The address is zero except for archive file members.
HEADER	Obtains the COFF file header structure.

Function Reference

This section describes the COFL functions alphabetically. Closely related functions are described together. For example, the `ldlinit` and `lditem` functions are grouped with `ldlread`.

ldahread

Reads an archive-member header

```
#include <stdio.h>
#include "ldfcn.h"

int ldahread (ldptr, arhead)
LDFILE *ldptr;
ARCHDR *arhead;
```

Discussion

To read an open archive header, use `ldahread`. The header of the archive currently associated with `ldptr` is put into memory beginning at `arhead`.

B

The `ldahread` function returns `SUCCESS` or `FAILURE`, defined in `ldfcn.h`. `FAILURE` indicates either:

- `TYPE(ldptr)` does not represent an archive file.
- The `ldahread` function cannot read the archive header.

ldclose, ldaclose

*Closes an object file or
an archive member*

```
#include <stdio.h>
#include "ldfcn.h"

int ldclose (ldptr)
LDFILE *ldptr;

int ldaclose (ldptr)
LDFILE *ldptr;
```

Discussion

For uniform access to both object files and archive members, use:

<code>ldclose</code>	for files opened with <code>ldopen</code>
<code>ldaclose</code>	for files opened with <code>ldaopen</code>

To close an archive member, keeping the archive open, use `ldclose`. The `ldclose` function returns `SUCCESS` or `FAILURE`, defined in `ldfcn.h`:

<code>FAILURE</code>	when <code>TYPE(ldptr)</code> represents an archive file and the archive contains more members. The <code>ldclose</code> function sets the offset of <code>ldptr</code> to the next member file address and prepares <code>ldptr</code> for a subsequent <code>ldopen</code> call.
----------------------	--

SUCCESS when the archive contains no more members, or when `TYPE(ldptr)` represents an individual object file.

To close an archive or object file regardless of the `TYPE(ldptr)`, use `ldaclose`. The `ldaclose` function:

- closes the file
- frees the memory allocated to the `LDFILE` structure associated with `ldptr`
- returns SUCCESS

ldfhread

Reads the file header

```
#include <stdio.h>
#include "ldfcn.h"

int ldahread (ldptr, filehead)
LDFILE *ldptr;
FILHDR *filehead;
```

Discussion

To read an open file header, use `ldfhread`. The object-file header associated with `ldptr` is put into memory beginning at `filehead`. The `ldfhread` function returns SUCCESS or FAILURE, defined in `ldfcn.h`, indicating whether the read operation is successful.

To access file-header fields without a function call, use `HEADER(ldptr)`, defined in `ldfcn.h`. The macro returns no value.

ldgetname

*Retrieves a name from
the object-file symbol
table*

```
#include <stdio.h>
#include "ldfcn.h"

char *ldgetname (ldptr, symbol)
LDFILE *ldptr;
SYMENT *symbol;
```

Discussion

To retrieve a name from the string table, use `ldgetname`. The returned address points to a static buffer local to `ldgetname`. To save the name, copy the static buffer, since the next `ldgetname` call overwrites the static buffer.

A `NULL` returned address indicates that the name cannot be retrieved, when:

- The string table cannot be found.
- The name is too long for the amount of memory allocated to the string table.
- The string table appears not to be a string table. For example, an auxiliary entry passed to `ldgetname` can apparently refer to a name in a nonexistent string table.
- The offset into the string table is beyond the end of the string table.

For example, you can call `ldgetname` immediately after a successful call to `ldtbread` to retrieve the name associated with the new symbol table entry.

ldlread, ldlnit, ldlitem

*Locates and reads the
function line-number
entries*

```
#include <stdio.h>
#include "ldfcn.h"

int ldlread(ldp_ptr, fcindex, linenum, linent)
LDFILE *ldp_ptr;
long fcindex;
unsigned short linenum;
LINENO *linent;

int ldlnit(ldp_ptr, fcindex)
LDFILE *ldp_ptr;
long fcindex;

int ldlitem(ldp_ptr, linenum, linent)
LDFILE *ldp_ptr;
unsigned short linenum;
LINENO *linent;
```

Discussion

To locate and read line-number entries:

<code>ldlread</code>	locates and reads a line-number entry for the function specified by the <code>fcindex</code> symbol-table entry.
<code>ldlnit</code>	locates the line-number entries for the specified function.
<code>ldlitem</code>	locates and reads a line-number entry for the current function.

B

Using `ldlinit` followed by `ldlitem` is the same as using `ldlread` alone. You can find the beginning of a series of line number entries with `ldlinit` or `ldlread`, then use `ldlitem` to retrieve the subsequent entries in the same function. For line number entries in a different function, call `ldlinit` or `ldlread` with a different `fcindex`.

You need not know an exact line number when calling `ldlread` or `ldlitem`. Both functions read the entry with the smallest line number equal to or greater than `linenum` into `linent`.

To specify the function for line number entry searches without reading a line number entry into `linent`, use `ldlinit`. To specify a new function and read a line number entry, use `ldlread`.

To find and read a line number entry without respecifying the function to be searched, use `ldlitem`.

The `ldlinit`, `ldlitem`, and `ldlread` functions return `SUCCESS` or `FAILURE`, defined in `ldfcn.h`. Failure can indicate:

Condition	Function
The object file contains no line number entries.	<code>ldlread</code> , <code>ldlinit</code>
The <code>fcnindx</code> matches no symbol table function entry.	<code>ldlread</code> , <code>ldlinit</code>
No line number is equal to or greater than <code>linenum</code> .	<code>ldlread</code> , <code>ldlitem</code>

ldlseek, ldnlseek

Seeks to the line-number entries of an object-file section

```
#include <stdio.h>
#include "ldfcn.h"

int ldlseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnlseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

Discussion

The `ldlseek` function seeks to the line-number entries of the section specified by `sectindx` of the COFF file currently associated with `ldptr`. The `ldnlseek` function seeks to the line-number entries of the section specified by `sectname`.

The `ldlseek` and `ldnlseek` functions return `SUCCESS` or `FAILURE`, defined in `ldfcn.h`:

- The `ldlseek` function fails when the variable `sectindx` is greater than the number of sections in the object file.
- The `ldnlseek` function fails when no section name corresponds with `*sectname`.
- Either function fails when the specified section has no line-number entries or when the function cannot seek to the specified entries.

The first section index is 1.

ldohseek

*Seeks to the COFF
optional file header*

```
#include <stdio.h>
#include "ldfcn.h"

int ldohseek (ldptr)
LDFILE *ldptr;
```

Discussion

The `ldohseek` function seeks to the optional file header of the COFF file currently associated with `ldptr`.

The `ldohseek` function returns `SUCCESS` or `FAILURE`, defined in `ldfcn.h`. Failure occurs when:

- The object file has no optional header.
- The function cannot seek to the optional header.

ldopen, ldaopen

*Opens an object file or
archive member for
reading*

```
#include <stdio.h>
#include "ldfcn.h"

LDFILE *ldopen (filename, ldptr)
char *filename;
LDFILE *ldptr;

LDFILE *ldaopen (filename, oldptr)
char *filename;
LDFILE *oldptr;
```

Discussion

The `ldopen` and `ldclose` functions provide uniform access to both simple object files and object files that are members of archive files.

When `ldptr` is `NULL`, `ldopen`:

- opens the file indicated by `filename`
- allocates and initializes the `LDFILE` structure
- returns a pointer to that structure to the calling program

When `ldptr` is valid and `TYPE(ldptr)` is an archive-file type number, `ldopen` reinitializes `LDFILE` for the next `filename` archive-file member.

Use `ldopen` and `ldclose` together. The `ldclose` function returns `FAILURE`, defined in `ldfcn.h`, only when `TYPE(ldptr)` is the archive magic number and the archive contains other members to be processed. In such cases, call `ldopen` with the current value of `ldptr`. In all other cases, especially when a new file is opened, call `ldopen` with a `NULL` `ldptr` argument.

For example:

```
/* for each file name to be processed */
ldptr = NULL;
do {
    if ( (ldptr = ldopen(filename, ldptr)) !=NULL ) {
        /* Check the file-type number. */
        /* Process the file. */
    }
} while (ldclose(ldptr) == FAILURE );
```

When `oldptr` is not `NULL`, `ldaopen`:

- opens `filename` as a new file
- allocates and initializes a new `LDFILE` structure
- copies the `TYPE`, `OFFSET`, and `HEADER` fields from `oldptr`

B

The `ldaopen` function returns a pointer to the new `LDFILE` structure, independent of `oldptr`. You can use both pointers concurrently to read separate parts of the object file. For example, use one pointer to step sequentially through the relocation information and the other to read indexed symbol-table entries.

Both `ldopen` and `ldaopen` open the specified file for reading and return `NULL` when the file cannot be opened or `LDFILE` structure memory cannot be allocated. A successful open operation does not ensure that the file is a COFF file or an archived object file.

ldrseek, ldnrseek

*Seeks to the file-section
relocation entries*

```
#include <stdio.h>
#include "ldfcn.h"

int ldrseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnrseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

Discussion

The `ldrseek` function seeks to the relocation entries of the section specified by the `sectindx` of the COFF file associated with `ldptr`.

The `ldnrseek` function seeks to the relocation entries of the section specified by `sectname`.

The `ldrseek` and `ldnrseek` functions return `SUCCESS` or `FAILURE`, as defined in `ldfcn.h`:

- The `ldrseek` function fails when `sectindx` is greater than the number of sections in the object file.
- The `ldnrseek` function fails when no section name corresponds with `*sectname`.
- Either function fails when the specified section has no relocation entries or when the function cannot seek to the specified relocation entries.

The first section index is 1.

ldshread, ldnshread

*Reads an indexed or
named file section
header*

```
#include <stdio.h>
#include "ldfcn.h"

int ldshread(ldptr, sectindx, secthead)
LDFILE *ldptr;
unsigned short sectindx;
SCNHDR *secthead;

int ldnshread(ldptr, sectname, secthead)
LDFILE *ldptr;
char *sectname;
SCNHDR *secthead;
```

Discussion

The `ldshread` function reads the section header specified by `sectindx` of the COFF file associated with `ldptr` into memory beginning at `secthead`. The `ldnshread` function reads the section header specified by `*sectname` into memory beginning at `secthead`.

B

The `ldshread` and `ldnshread` functions return `SUCCESS` or `FAILURE`, defined in `ldfcn.h`:

- The `ldshread` function fails when `sectindx` is greater than the number of sections in the object file.
- The `ldnshread` function fails when no section name corresponds with `*sectname`.
- Either function fails when it cannot read the specified section header.

The first section-header index is 1.

ldsseek, ldnsseek

*Seek to an indexed or
named file section*

```
#include <stdio.h>
#include "ldfcn.h"

int ldsseek (ldptr, sectindx)
LDFILE *ldptr;
unsigned short sectindx;

int ldnsseek (ldptr, sectname)
LDFILE *ldptr;
char *sectname;
```

Discussion

The `ldsseek` function seeks to the section specified by `sectindx` of the common object file currently associated with `ldptr`.

The `ldnsseek` function seeks to the section specified by `*sectname`.

The `ldsseek` and `ldnsseek` functions return `SUCCESS` or `FAILURE`, defined in `ldfcn.h`:

- The `ldsseek` function fails when the variable `sectindx` is greater than the number of sections in the object file.
- The `ldnsseek` function fails when no section name corresponds with `*sectname`.
- Either function fails when the specified section has no section data or when the function cannot seek to the section data.

The first section index is 1.

ldtbindex

Computes the symbol-table-entry index

```
#include <stdio.h>
#include "ldfcn.h"

long ldtbindex (ldptr)
LDFILE *ldptr;
```

Discussion

The `ldtbindex` function returns the index of the symbol-table entry at the current position of the COFF file associated with `ldptr`. The index is a long integer.

You can use the index in subsequent calls to `ldtbread`. Calling `ldtbindex` immediately after reading a particular symbol-table entry returns the next entry index, because `ldtbindex` returns the index of the symbol-table entry that begins at the current position of the object file.

B

The `ldtbindex` function fails when the object file contains no symbols or when the object file is not positioned at the beginning of a symbol-table entry.

The first symbol index in the symbol table is 0.

ldtbread

*Reads an indexed
symbol-table entry*

```
#include <stdio.h>
#include "ldfcn.h"

int ldtbread (ldptr, symindex, symbol)
LDFILE *ldptr;
long symindex;
SYMENT *symbol;
```

Discussion

The `ldtbread` function reads the `symindex` symbol-table entry of the COFF file associated with `ldptr` into memory beginning at `symbol`.

The `ldtbread` function returns `SUCCESS` or `FAILURE`, defined in `ldfcn.h`. Failure occurs when `symindex` is greater than the number of symbols in the object file or when `ldtbread` cannot read the symbol-table entry.

ldtbseek

Seeks to the symbol table

```
#include <stdio.h>
#include "ldfcn.h"

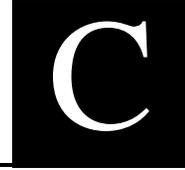
int ldtbseek (ldptr)
LDFILE *ldptr;
```

Discussion

The `ldtbseek` function seeks to the symbol table of the object file associated with `ldptr`.

The `ldtbseek` function returns `SUCCESS` or `FAILURE`, defined in `ldfcn.h`. The `ldtbseek` function fails when the symbol table has been stripped from the object file or when the function cannot seek to the symbol table.

Common Object File Format (COFF) and Common Archive File Format (CAFF)



This chapter describes the i960® processor common object file format (COFF) and the associated common archive file format (CAFF) standards.

Characteristics of COFF

COFF applies to two kinds of files: relocatable binary files and executable files. Relocatable binary files are produced by the assembler and by some linker options. Executable files are created from relocatable binary files by the linker.

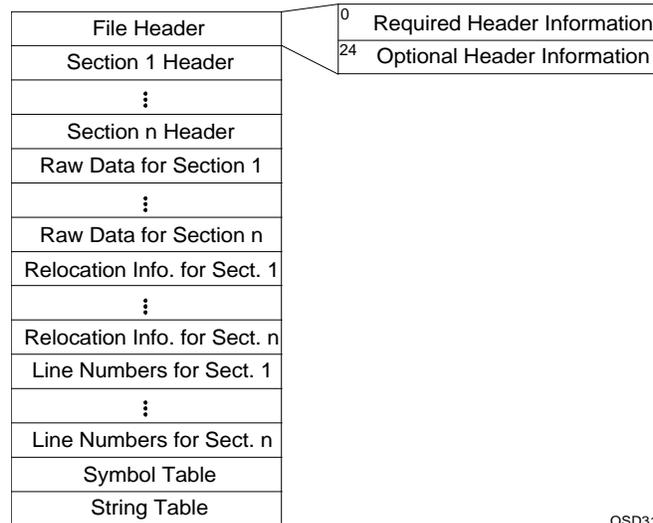
Figure C-1 shows the relation of headers to the information in COFF.



NOTE. *The only optional header the assembler uses is the execution header.*

When you link a program with the linker's strip option, relocation information, line numbers, the symbol table, and the string table are deleted. Or you can remove line number information, the symbol table, and the string table with the stripper.

Line numbers do not appear unless the program is compiled with the compiler's debug control. If all external references are resolved at link-time, no relocation information is included. The string table is also omitted when the source file contains no symbols with names longer than eight characters.

Figure C-1 Object File Format

Definitions and Conventions

Be sure you are familiar with these definitions and conventions before using COFF. You can find additional terms in the glossary.

Sections

A section is the smallest portion of an object file that can be relocated and treated as a separate entity by the linker. By default, an object file has three sections that are loaded into memory when the file is executed. The sections and their contents are:

.text	the executable code for each instruction
.data	initialized data variables
.bss	uninitialized data variables

Additional sections can accommodate multiple text or data blocks, shared data blocks, or user-specified sections.

For a linked file, each COFF section in that file has a begin and an end symbol. Generally, `__Bname` is the begin symbol for each section and `__Ename` is the end symbol, where *name* matches the COFF section name. The begin and end symbols are limited to a length of 8 characters.

Physical and Virtual Address

In most COFF files, the physical address and virtual address of each section or symbol are the same, even though the section heading contains an address field for both. For example, on a system with paging, the address is located relative to address zero of virtual memory and the operating system performs another address translation.

File Header

The file header contains 20 bytes of information about the object file. Table C-1 shows the contents of the file.

The `f_opthdr` field contains the size of the optional header information. The i960® processor utilities, such as the dumper, work properly on any common object file because they use the contents of the `f_opthdr` field to locate the end of the optional header information and seek past the header.

C

Table C-1 File Header Contents

Bytes	Declaration	Name	Description
0-1	unsigned short	f_magic	file type number
2-3	unsigned short	f_nscns	number of section headers (equals the number of sections)
4-7	long int	f_timdat	time and date stamp indicating when the file was created relative to the number of elapsed seconds since 00:00:00 GMT, January 1, 1970
8-11	long int	f_symptr	file pointer containing the starting address of the symbol table
12-15	long int	f_nsyms	number of entries in the symbol table
16-17	unsigned short	f_opthdr	number of bytes in the optional header
18-19	unsigned short	f_flags	flags

File Header Declaration

The C structure declaration for the file header is in the `coffcoff.h` header file. Example C-1 shows the declaration format.

Example C-1 File Header Declaration

```

struct filehdr {
    unsigned short  f_magic;    /* magic number */
    unsigned short  f_nscns;   /* number of section */
    long            f_timdat;   /* and date stamp */
    long            f_symptr;   /* ptr to symbol table */
    long            f_nsyms;   /* number entries in
                               symbol table */
    unsigned short  f_opthdr;   /* size of optional
                               header */
    unsigned short  f_flags;   /* flags */
};

#define FILHDR struct filehdr
#define FILHSZ sizeof(FILHDR)

```

File Header Flags

The file header flags describe the type of the object file. Table C-2 provides definitions of the flags.

Table C-2 File Header Flags

Mnemonic	Definition
F_RELFLG	indicates whether relocation information was stripped out.
F_EXEC	set if the file is executable and has no unresolved external references.
F_LNNO	set if line numbers were stripped out.
F_LSYMS	set if local symbols were stripped out.
F_AR32WR	set if byte ordering is little-endian.
F_AR32W	set if byte ordering is big-endian.
F_PIC	set if the file contains position-independent code.

continued 

Table C-2 File Header Flags (continued)

Mnemonic	Definition
F_PID	set if the file contains position-independent data.
F_LINKPID	set if the file is suitable for linking with position-independent code or data.
F_BIG_ENDIAN_T	set if target information is in big-endian byte order.
F_SECT_SYM	set in symbols representing section names.

The upper four bits of the flag's word contains the architecture type. Table C-3 lists the flag names.

Table C-3 Architecture Types of File Header Flags

Mnemonic	Definition
F_I960CORE	architecture common to all i960 processors
F_I960KB or F_I960SB	architecture common to KB and SB processors
F_I960XA	architecture common to KA, SA, and CA processors
F_I960CA	architecture common to CA and CF processors
F_I960KA or F_I960SA	architecture common to KA and SA processors

File Type Numbers

In the file header, the first two bytes indicate the target on which the object file can be executed. These file type numbers are defined as follows:

```
#define I960ROMAGIC 0x0160 /* read-only text segments */
#define I960RWMAGIC 0x0161 /* read-write text segments */
```

Execution File Header Declaration

The execution file header is the first data structure in the optional file header that immediately follows the required header information. See Table C-1 for the location and size of execution information in the file header.

The C language structure declaration for the i960® processor-executable (`a.out`) file header is in the `coff.h` header file. Refer to Table C-4 for the declaration's fields.

Table C-4 Standard Output (`a.out`) File Header

Bytes	Declaration	Name	Description
0-1	short	<code>f_magic</code>	magic number
2-3	short	<code>vstamp</code>	version stamp
4-7	unsigned long	<code>tsize</code>	text size in bytes, padded to full word boundary
8-11	unsigned long	<code>dsize</code>	initialized data size
12-15	unsigned long	<code>bsize</code>	uninitialized data size
16-19	unsigned long	<code>entry</code>	entry point
20-23	unsigned long	<code>text_start</code>	base of text for this file
24-27	unsigned long	<code>data_start</code>	base of data for this file
28-31	unsigned long	<code>tagentries</code>	number of tag entries to follow

The `tagentries` field is always zero because none of Intel's development tools use tag entries.

Section Headers

A table of section headers specifies the layout of data within the file. Table C-5 shows the section header format. The size of a section is padded to a multiple of 4 bytes.

Table C-5 Section Header Contents

Bytes	Declaration	Name	Description
0-7	char	s_name	8-character section name, padded with zeros
8-11	long int	s_paddr	physical address of section
12-15	long int	s_vaddr	virtual address of section
16-19	long int	s_size	section size in bytes
20-23	long int	s_scnptr	file pointer to raw data
24-27	long int	s_relptr	file pointer to relocation entries
28-31	long int	s_lnnoptr	file pointer to line number entries
32-33	unsigned short	s_nreloc	number of relocation entries
34-35	unsigned short	s_nlnno	number of line entries
36-39	long int	s_flags	flags
40-43	unsigned long int	s_align	alignment of the section to the specified byte boundary



NOTE. *The Intel 80960 assembler rounds section sizes to the next higher 4-byte word boundary.*

Section Header Declaration

The C structure declaration for the section headers is in the `coff.h` header file. Example C-2 shows the declaration format.

Example C-2 Section Header Declaration

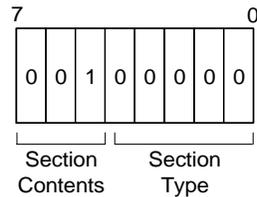
```
struct scnhdr
{
    char          s_name[8]; /* section name */
    long          s_paddr;   /* physical address */
    long          s_vaddr;   /* virtual address */
    long          s_size;    /* section size */
    long          s_scnptr;  /* file ptr to section
                             raw data */
    long          s_relptr;  /* file ptr to
                             relocation */
    long          s_lnnoptr; /* file ptr to line
                             number */
    unsigned short s_nreloc; /* number of
                             relocation entries */
    unsigned short s_nlnno;  /* number of line
                             entries */
    long          s_flags;   /* flags */
    unsigned long  s_align;  /* section alignment */
};
#define SCNHDR  struct scnhdr
#define SCNHSZ  sizeof(SCNHDR)
```

Section Header Flags

Section header flags indicate the section type. Table C-6 shows the flag format.

The flag field occupies one byte. The value in the first five bits of the byte indicates the section type. The value in the last three bits indicates the contents of the section. Figure C-2 shows the flag field for a regular text section.

Figure C-2 Flag Field Values



OSD1134

Table C-6 Section Header Flags

Mnemonic	Flag	Definition
STYP_REG	0x00	regular section (allocated, relocated, loaded)
STYP_DSECT	0x01	dummy section (not allocated, relocated, not loaded)
STYP_NOLOAD	0x02	noload section (allocated, relocated, not loaded)
STYP_GROUP	0x04	grouped section (formed from input sections)
STYP_PAD	0x08	padding section (not allocated, not relocated, loaded)
STYP_COPY	0x10	copy section (for a decision function used in updating fields; not allocated, not relocated, loaded, relocation and line number entries processed normally)
STYP_TEXT	0x20	section contains executable text
STYP_DATA	0x40	section contains initialized data
STYP_BSS	0x80	section contains uninitialized data

Sections

The raw data for each section follows the list of section headers. If a section has data, the `s_scnptr` field in its section header points to it (see Figure C-2). For a section with no raw data (a `bss` section, for example), the `s_scnptr` field contains a null value.

Relocation

A relocation entry is used by the linker. It consists of the address at which relocation should occur, the index of the relevant symbol in the symbol table, and the type of relocation required. Table C-7 shows relocation entry format.

Table C-7 Relocation Entry Format

Bytes	Declaration	Name	Description
0-3	long int	<code>r_vaddr</code>	address of reference
4-7	long int	<code>r_symndx</code>	symbol table index
8-9	unsigned short	<code>r_type</code>	relocation type
10-11	char	<code>padder[2]</code>	padding---not used

Table C-8 contains relocation types defined for the i960® processor.

Table C-8 Relocation Types

Mnemonic	Decimal Value	Hexadecimal Value	Definition
R_RELLONG	17	0x0011	direct 32-bit relocation
R_RELSHORT	22	0x0016	direct 12-bit relocation
R_IPRSHORT	24	0x0018	unimplemented
R_IPRMED	25	0x0019	IP-relative relocation
R_IPRLONG	26	0x001A	32-bit IP-relative relocation
R_OPTCALL	27	0x001B	optimizable call (callj)
R_OPTCALLX	28	0x001C	optimizable call (calljx)
R_GETSEG	29	0x001D	unimplemented
R_GETPA	30	0x001E	unimplemented
R_TAGWORD	31	0x001F	unimplemented

Relocation Entry Declaration

The structure declaration for relocation entries is in the `coff.h` header file. Refer to Example C-3 for the declaration format.

Example C-3 Relocation Entry Declaration

```

struct reloc
{
    long          r_vaddr;    /* virtual address of
                             reference */
    long          r_symndx;   /* symbol table index
                             into symbol table */
    unsigned short r_type;   /* relocation type */
    char          padder[2]  /* padding */
};

#define RELOC      struct reloc
#define RELSZ     sizeof(RELOC)

```

Direct Relocation

In direct relocation, the linker adds the 32-bit address of the requested symbol to the value at a given location. In this example, the assembler places the value 4 at location `x` and issues a `R_RELLONG` request for `fumble`. At link-time, linker adds the value of `fumble` to the word at `x`.

```
.glob          fumble
x:             .word    fumble+4
```

IP-relative Relocation

In IP-relative relocation, the linker adds the value of the following expression to the offset in the instruction specified by the relocation entry.

$$\text{addr}(\text{symbol}) - (\text{base address of current segment})$$

The branch instruction takes a 24-bit IP-relative offset. In the following example, the assembler places the negation of the current location counter in the branch instructions offset field (in this example, `0xfffff0`) and issues an `r_iprmed` relocation request for this branch instruction. At link-time, the linker adds the value of `fumble` and subtracts the value of `tstart` from the branch instruction's offset. The result is the true IP-relative offset of `fumble`. The branch address must be within 24 bits.

```
.globl fumble
.text
tstart:
.space 0x10
x:  b fumble
/* disassembly for r.0 */
/* section      .text */
    .text
    0: 00000000                .word 0x0
    4: 00000000                .word 0x0
    8: 00000000                .word 0x0
    c: 00000000                .word 0x0
   10: 08fffff0                b      0x0
```

```

***RELOCATION INFORMATION***
      Vaddr      Symndx      Type      Name
r.0:
.text:
      0x00000010          10  IPRMED      fumble
.data:
.bss:

```

Line Number Entry

Invoke the compiler with the debug option to get a listing of line numbers where you can place breakpoints to make debugging easier. All line numbers in a section are grouped by function, as shown in Example C-4, and are relative to the beginning of a function. The `s_lnnoptr` field in the section header points to the first line number entry for that section.

Example C-4 Line Number Grouping

```

symbol index          0
physical address      line number
physical address      line number
:                     :
.                     .
symbol index          0
physical address      line number
physical address      line number

```

The first entry for each section has line number zero and contains the symbol table index of the function name. Each following entry associates each line number with the address of the code generated for it.

The structure declaration for line number entries is in the `coff.h` header file. Example C-5 shows the structure declaration for line number entries.

Example C-5 Line Number Entry Declaration

```
struct lineno
{
    union
    {
        long      l_symndx; /* symbol table index
                           of func name*/
        long      l_paddr; /* paddr of line
                           number*/
    } l_addr;
    unsigned short l_lnno; /* line number */
    char          padding[2] /* not used */
};

#define LINENO      struct lineno
#define LINESZ      sizeof(LINENO)
```

Symbol Table

The symbol table consists of at least one fixed-length entry per symbol, with some symbols followed by auxiliary entries. Each entry includes the value, the type, and other information. Example C-6 shows the order in which symbols are listed. `f_symstr` in the file header points to the beginning of the symbol table. The `f_nsyms` field in the file header contains the total number of entries in the symbol table.

Example C-6 COFF Symbol Table

```
file name 1
function 1
local symbols for function 1
function 2
local symbols for function 2
.
.
.
static variables
.
.
.
file name 2
function 1
local symbols for function 1
.
.
.
static variables
.
.
.
defined global symbols
undefined global symbols
```

Symbol Table Entries

The symbol table consists of two general kinds of entries, each 24 bytes long. The first type is the main entry, representing a symbol. The format of the main entry is shown in Table C-9. The second type of entry is an auxiliary entry, whose format varies depending on how it is used.

Table C-9 Symbol Table Entry Format

Bytes	Declaration	Name	Definition
0-7		<code>_n</code>	the name of a pointer or of a symbol
8-11	<code>long int</code>	<code>n_value</code>	symbol value; storage class dependent
12-13	<code>short</code>	<code>n_scnum</code>	section number of symbol
14-15	<code>unsigned short</code>	<code>n_flags</code>	pic, pid flags for the module containing the symbol, flag when symbol is a section name
16-19	<code>unsigned long</code>	<code>n_type</code>	basic and derived type specification
20	<code>char</code>	<code>n_sclass</code>	storage class
21	<code>char</code>	<code>n_numaux</code>	number of auxiliary entries
22-23	<code>char</code>	<code>pad2[2]</code>	padding to force alignment

Structure for Symbol Table Entries

Example C-7 shows the C language structure declaration for the symbol table entry that can be found in the `coff.h` header file.

Example C-7 Symbol Table Entry Declaration

```
#define SYMNMLEN 8          /* Number of characters in a
                             symbol name */
struct syment {
    union {
        char    _n_name[SYMNMLEN];      /* symbol name */
        struct {
            long  _n_zeroes; /* zero - name in string table */
            long  _n_offset; /* offset into string table */
        } _n_n;
        char    *_n_nptr[2]; /* allows for overlaying */
    } _n;
    long    n_value; /* value of symbol */
    short   n_scnum; /* section number */
}
```

continued 

Example C-7 Symbol Table Entry Declaration (continued)

```

        unsigned short n_flags;    /* copy of "flags" from */
                                   /* input file header if */
                                   /* not a section symbol.*/
        unsigned long   n_type;    /* type and derived type */
        char            n_sclass; /* storage class */
        char            n_numaux; /* number of aux. entries */
        char            pad2[2];   /* force alignment */
};

#define n_name      _n._n_name
#define n_nptr     _n._n_nptr[1]
#define n_zeroes   _n._n_n._n_zeroes
#define n_offset   _n._n_n._n_offset

.
.
.

#define SYMENT      struct syment
#define SYMESZ     sizeof(SYMENT)

```

Symbols and Inner Blocks .bb/.eb

A block is a compound statement, and an inner block is a block that occurs within a function that is itself a block.

For each inner block that uses local symbols, the symbol `.bb` appears in the symbol table right before the first local symbol of that block, and the symbol `.eb` appears right after the last local symbol. The sequence is shown here:

```

.bb
local symbols for that block
.eb

```

Example C-8 is a C language example that shows nesting the `.bb` and `.eb` pair and associated symbols.

Example C-8 Nested Blocks

```

{                                     /* block 1 */
    int i;
    char c;
    ...
    {                                     /* block 2 */
        long a;
        ...
        {                                     /* block 3 */
            int x;
            ...
        }                                     /* block 3 */
    }                                     /* block 2 */
    {                                     /* block 4 */
        long i;
        ...
    }                                     /* block 4 */
}                                     /* block 1 */

```

Example C-9 shows the format of the symbol table for these nested blocks.

Example C-9 Example of a Symbol Table

```

.bb for block 1
    i
    c
.bb for block 2
    a
.bb for block 3
    x
.eb for block 3
.eb for block 2
.bb for block 4
    i
.bb for block 4
.eb for block 1

```

Symbols and Functions `.bf/.ef, .target`

In the symbol table, the symbol `.bf` appears between the function name and the first local symbol of the function, and the symbol `.ef` appears right after the last local symbol. The sequence is shown here:

```
function name
    .bf
    local symbol
    .ef
```

When the return value of the function is a structure or union, the compiler creates a `.target` symbol for storing the function-return. This symbol is an automatic variable of the type `pointer` and has a value field of zero. It appears in the symbol table between the function name and the symbol `.bf`, as shown here:

```
function name
    .target
    .bf
local symbols
    .ef
```

Special Symbols

The symbol table contains special symbols that are generated by the assembler, the compiler, and other utilities. Table C-10 shows a list of these symbols.

Table C-10 Special Symbols in the Symbol Table

Symbol	Definition
.file	filename
.text	address of .text section
.data	address of .data section
.bss	address of .bss section
.bb	address of start of inner block
.eb	address of end of inner block
.bf	address of start of function
.ef	address of end of function
.target	pointer to the structure or union returned by a function
.xfake	dummy tag name for structure, union, or enumeration
.eos	end of members of structure, union, or enumeration
_etext	next available address after the output section .text
_edata	next available address after the output section .data
_end	next available address after the output section .bss
__Bname	address of beginning of <i>name</i>
__Ename	address of end of <i>name</i>

The symbols `__Bname` and `__Ename` are generated by the linker as a convenience to the user. `__Bname` marks the beginning of a section denoted as *name* and `__Ename` marks the byte following the last byte of the section.



NOTE. These symbol names are preceded by a double underscore (`__`). These symbol names cannot exceed 8 characters in length.

When `__Bname` and `__Ename` mark the beginning and end of the `.text`, `.data`, and `.bss` sections, the initial period in the filenames is dropped. Thus, the sections `.text` and `.data` would be delimited by `__Btext`, `__Etext`, `__Bdata`, and `__Edata`. The initial period in any user-defined section, however, is retained. A user file called `.mysec`, for example, would be delimited by `__B.mysec` and `__E.mysec`.

Six special symbols come in pairs. The `.bb` and `.eb` pair indicates the boundaries of inner blocks. The `.bf` and `.ef` pair brackets each function. The `.xfake` and `.eos` pair names and defines the limit of unnamed structures, unions, and enumerations; The `.eos` symbol also appears after named structures, unions, and enumerations.

When a structure, union, or enumeration has no tag name, the compiler invents a symbol table name: `.xfake;`, where `x` is an integer greater than zero. Several unnamed items are tagged consecutively, as follows: `.1fake`, `.2fake`, `.3fake` ... `.11fake`, `.12fake`, etc.

For each section the assembler creates, it generates a symbol table entry in which bit 0x200 in the symbol flags field is set. This creates a signature recognized by the tools that process COFF files.

The linker does not generate symbols for the sections it creates.

Symbol Name

The symbol name structure is actually a union declared like this:

```
union {
    char n_name[8];
    struct string_table_pointer {
        long n_zeroes;
        long n_offset;
    };
};
```

If a symbol name is less than eight characters long, it is stored in `n_name`, padded if necessary with nulls. If the symbol name is longer than 8 characters, `n_zeroes` is set to 0 and `n_offset` is set to the offset into the string table at which the symbol name is stored.

Table C-11 Symbol Name Field

Bytes	Declaration	Name	Description
0-7	char	n_name	null-padded symbol name
0-3	long_int	n_zeros	zero in this field indicates that the name is in the string table
4-7	long_int	n_offset	offset of the name in the string table

Storage Classes

Storage class is associated with each symbol by the compiler or assembler and stored in the `n_sclass` field. However, the following are used only externally: `C_EFCN`, `C_EXTDEF`, `C_ULABEL`, `C_USTATIC`, and `C_LINE`. Table C-12 provides definitions of the storage classes.

Table C-12 Storage Classes

Mnemonic	Decimal Value	Hexadecimal Value	Storage Class
<code>C_EFCN</code>	-1		physical end of a
<code>C_NULL</code>	0	0x0000	unknown
<code>C_AUTO</code>	1	0x0001	automatic variable
<code>C_EXT</code>	2	0x0002	external symbol
<code>C_STAT</code>	3	0x0003	static variable
<code>C_REG</code>	4	0x0004	register variable
<code>C_EXTDEF</code>	5	0x0005	external definition
<code>C_LABEL</code>	6	0x0006	label
<code>C_ULABEL</code>	7	0x0007	undefined
<code>C_MOS</code>	8	0x0008	member of structure
<code>C_ARG</code>	9	0x0009	function argument in an argument block
<code>C_STRTAG</code>	10	0x000A	structure tag

continued 

C

Table C-12 Storage Classes (continued)

Mnemonic	Decimal Value	Hexadecimal Value	Storage Class
C_MOU	11	0x000B	member of union
C_UNTAG	12	0x000C	union tag
C_TPDEF	13	0x000D	type definition
C_USTATIC	14	0x000E	uninitialized static variable
C_ENTAG	15	0x000F	enumeration tag
C_MOE	16	0x0010	member of enumeration
C_REGPARM	17	0x0011	function argument in a register
C_FIELD	18	0x0012	bit field
C_AUTOARG	19	0x0013	function argument in the callee's frame
C_BLOCK	100	0x0064	beginning and end of
C_FCN	101	0x0065	beginning and end of
C_EOS	102	0x0066	end of
C_FILE	103	0x0067	filename
C_LINE	104	0x0068	used only by utility programs
C_ALIAS	105	0x0069	duplicated tag
C_HIDDEN	106	0x006A	used to avoid name conflicts
C_SCALL	107	0x006B	reached by a system call
C_LEAFEXT	108	0x006C	global leaf procedure: can be called with BAL
C_LEAFSTAT	113	0x0071	static leaf procedure: can be called with BAL

Storage Classes for Special Symbols

Restricted storage classes, used only for certain symbols, are given in Table C-13.

Table C-13 Restricted Storage Classes

Storage Class	Special Symbol
C_BLOCK	.bb, .eb
C_FCN	.bf, .ef
C_EOS	.eos
C_FILE	.file

Call Optimization

Call optimization occurs when the linker matches an `R_OPTCALL` relocation request with a symbol of storage class `C_SCALL`, `C_LEAFSTAT`, or `C_LEAFEXT`. When the storage class is `C_SCALL`, the linker replaces the `call` instruction with a `calls` instruction to the appropriate system procedure index. For the `C_LEAFEXT` or `C_LEAFSTAT` storage classes, the linker replaces the `call` with a branch-and-link instruction to a special entry point in the destination procedure.

`C_SCALL` is the storage class associated with names of routines that can be called with the `calls` instruction. Symbols of type `C_SCALL` have two auxiliary entries, the second of which contains the index of the destination procedure in a table of system calls.

The `C_LEAFEXT` storage class is associated with routines that can be called with the branch-and-link (`bal`) instruction. Such routines can have two entry points. The address of the first, the `call` entry, is given as the value of the routine name and supports access via a `call` instruction. The address of the second is contained in the second auxiliary entry.

C

The `C_LEAFSTAT` storage class also is associated with routines to be called with `ba1`, but the assembler optimizes the functions instead of the linker because the routine is of source module scope.

Symbol Value Field

The value of a symbol depends on its storage class. This relationship is summarized in Table C-14.

Table C-14 Symbol Value Field

Storage Class	Decimal Value	Hexadecimal Value	Argument Value
<code>C_AUTO</code>	1	0x0001	frame pointer offset in bytes
<code>C_EXT</code>	2	0x0002	relocatable
<code>C_STAT</code>	3	0x0003	relocatable address
<code>C_REG</code>	4	0x0004	register number: r0 = 0,...,r15 = 15, g0 = 16,...,g15 = 31
<code>C_LABEL</code>	6	0x0006	relocatable address
<code>C_MOS</code>	7	0x0008	offset in bytes
<code>C_ARG</code>	8	0x0009	argument block offset
<code>C_STRTAG</code>	9	0x000A	zero
<code>C_MOU</code>	11	0x000B	zero
<code>C_UNTAG</code>	12	0x000C	zero
<code>C_TPDEF</code>	13	0x000D	zero
<code>C_ENTAG</code>	15	0x000F	zero
<code>C_MOE</code>	16	0x0010	enumeration value
<code>C_REGPARAM</code>	17	0x0011	register number r0 = 0,...,r15 = 15, g0 = 16,...,g15 = 31
<code>C_FIELD</code>	18	0x0012	bit displacement
<code>C_AUTOARG</code>	19	0x0013	frame pointer offset in bytes

continued ➡

Table C-14 Symbol Value Field (continued)

Storage Class	Decimal Value	Hexadecimal Value	Argument Value
C_BLOCK	100	0x0064	relocatable address
C_FCN	101	0x0065	relocatable address
C_EOS	102	0x0066	size
C_FILE	103	0x0067	symbol table entry index for next .file symbol
C_ALIAS	105	0x0069	tag index
C_HIDDEN	106	0x006A	relocatable address

A symbol with storage class `C_FILE` has a value that points to the next `.file` symbol in the symbol table, or the beginning of the global symbols in the case of the last `.file` symbol. Before files are linked, the value of the `.file` symbol is zero.

Relocatable address symbols have a value equal to the address of the symbol. When the linker relocates the section, the value of the symbol changes.

Section Number Field

The section number field indicates the section in which a symbol is defined. Table C-15 shows the defined constants used to refer to this field.

Table C-15 Section Number Field

Symbol Name	Section Number	Definition
N_DEBUG	-2	symbolic debugging symbol, including tag names for structures, unions, or enumerations, typedefs, and name of file
N_ABS	-1	absolute symbol, not relocatable
N_UNDEF	0	undefined external symbol
N_SCNUM	any integer > 0	section number in which symbol is defined

Absolute symbols include automatic and register variables, function arguments, and `.eos` symbols. The `.text`, `.data`, and `.bss` symbols default to section numbers 1, 2, and 3 respectively.

A section number of zero indicates a relocatable external symbol that is undefined in the current file. However, a multiply-defined external symbol (i.e., an uninitialized variable defined outside a function in C) has a section number of zero and a positive value, which gives the symbol size.

When files with multiply defined external symbols are combined, the linker combines all input symbols into one symbol and assigns the `.bss` section number. The size of the combined symbols determines how much memory is allocated, and the value becomes the address of the symbol.

Section Numbers and Storage Classes

Symbols with certain storage classes are restricted to certain section numbers. This relationship is summarized in Table C-16.

Table C-16 Section Number and Storage Class

Storage Class	Section Number	Hexadecimal Value	Symbol Names
C_AUTO	-1	0x0001	N_ABS
C_EXT	-1, 0, 1 to 077777	0x0002	N_ABS, N_UNDEF, N_SCNUM
C_STAT	1 to 077777	0x0003	N_SCNUM
C_REG	-1	0x0004	N_ABS
C_LABEL	-1, 0, 1 to 077777	0x0006	N_UNDEF, N_SCNUM, N_ABS
C_MOS	-1	0x0008	N_ABS
C_ARG	-1	0x0009	N_ABS
C_STRTAG	-2	0x000A	N_DEBUG
C_MOU	-1	0x000B	N_ABS
C_UNTAG	-2	0x000C	N_DEBUG
C_TPDEF	-2	0x000D	N_DEBUG
C_ENTAG	-2	0x000F	N_DEBUG
C_MOE	-1	0x0010	N_ABS
C_REGPARM	-1	0x0011	N_ABS
C_FIELD	-1	0x0012	N_ABS
C_BLOCK	1 to 077777	0x0064	N_SCNUM
C_FCN	1 to 077777	0x0065	N_SCNUM
C_EOS	-1	0x0066	N_ABS
C_FILE	-2	0x0067	N_DEBUG
C_ALIAS	-2	0x0069	N_DEBUG

Type Entry

The type field in the symbol table entry contains information about the basic and derived type for the symbol. The compiler generates this information when the debug option is used. Each symbol has one basic or fundamental type but can have more than one derived type.

The format of the 32-bit type entry is:

```
d13 ... d6   d5   d4   d3   d2   d1   type
```

Bit order is from bit 31 on the left to bit 0 on the right. Bits 4 through 0, indicated above by *type*, specify one of the fundamental types given in Table C-17. Fundamental types are determined by the user input type. Bits 5 through 30 are arranged as thirteen 2-bit fields referred to as *d1* through *d13*. These fields represent levels of the derived types with the values shown in Table C-18.

Two examples demonstrate the interpretation of the symbol table entry representing *type*.

In the first example, the function `func` returns a pointer to a character.

```
char *func();
```

The fundamental type is 2 (character), the *d1* field is 2 (function), and the *d2* field is 1 (pointer). Therefore, the type word in the symbol table for `func` contains the hexadecimal number `0xC2`, indicating a function that returns a pointer to a character.

In the second example, the `tabptr` identifier is a three-dimensional array of pointers to short integers.

```
short *tabptr[10][25][3];
```

The fundamental type of `tabptr` is 3 (short integer); the *d1*, *d2*, and *d3* fields each contain a 3 (array), and the *d4* field is 1 (pointer). Therefore, the type entry in the symbol table contains the hexadecimal number `0xFE3`, indicating a three-dimensional array of pointers to short integers.

Table C-17 Fundamental Types

Mnemonic	Decimal Value	Hexadecimal Value	Definition
T_NULL	0	0x0000	not assigned
T_VOID	1	0x0001	void
T_CHAR	2	0x0002	character
T_SHORT	3	0x0003	short integer
T_INT	4	0x0004	integer
T_LONG	5	0x0005	long integer
T_FLOAT	6	0x0006	floating point
T_DOUBLE	7	0x0007	double word
T_STRUCT	8	0x0008	structure
T_UNION	9	0x0009	union
T_ENUM	10	0x000A	enumeration
T_MOE	11	0x000B	member of enumeration
T_UCHAR	12	0x000C	unsigned character
T_USHORT	13	0x000D	unsigned short
T_UINT	14	0x000E	unsigned integer
T_ULONG	15	0x000F	unsigned long
T_LNGDBL	16	0x0010	long double

Table C-18 Derived Types Field Values

Mnemonic	Decimal Value	Hexadecimal Value	Definition
DT_NON	0	0x00000	no derived type
DT_PTR	1	0x0001	pointer
DT_FCN	2	0x0002	function
DT_ARY	3	0x0003	array

Type Entries and Storage Classes

Table C-19 shows the derived type entries that are legal for each storage class.

Table C-19 Type Entries by Storage Class

Storage Class	-----Derived Type Entries-----			Basic Type
	Function	Array	Pointer	
C_AUTO	no	yes	yes	any except T_MOE
C_EXT	yes	yes	yes	any except T_MOE
C_STAT	yes	yes	yes	any except T_MOE
C_REG	no	no	yes	any except T_MOE
C_LABEL	no	no	no	T_NULL
C_MOS	no	yes	yes	any except T_MOE
C_ARG	yes	no	yes	any except T_MOE
C_STRTAG	no	no	no	T_STRUCT
C_MOU	no	yes	yes	any except T_MOE
C_UNTAG	no	no	no	T_UNION
C_TPDEF	no	yes	yes	any except T_MOE
C_ENTAG	no	no	no	T_ENUM
C_MOE	no	no	no	T_MOE
C_REGPARM	no	no	yes	any except T_MOE
C_FIELD	no	no	no	T_ENUM T_UCHAR T_USHORT T_UNIT T_ULONG
C_BLOCK	no	no	no	T_NULL
C_FCN	no	no	no	T_NULL

continued 

Table C-19 Type Entries by Storage Class (continued)

Storage Class	-----Derived Type Entries-----			Basic Type
	Function	Array	Pointer	
C_EOS	no	no	no	T_NULL
C_FILE	no	no	no	T_NULL
C_ALIAS	no	no	no	T_STRUCT T_UNION T_ENUM

Conditions for the *d* entries apply to *d*1 through *d*13, except that you cannot have two consecutive derived types of function, that is, you cannot have a function returning a function.

Although function arguments can be declared as arrays, the compiler changes them to pointers by default. Therefore, a function argument cannot have array as its first derived type.

Auxiliary Table Entries

The auxiliary table entry or entries for a symbol have 24 bytes each. Every symbol has an auxiliary table entry with the same number of bytes as the symbol has in the symbol table entry. Table C-20 provides a summary of the auxiliary symbol table format. The formats are discussed in detail in subsequent sections.

C

Table C-20 Auxiliary Symbol Table Entries

Name	Storage Class	---Type of Entry---		Auxiliary Entry Format
		d1	typ	
.file	C_FILE	DT_NON	T_NULL	filename - possibly followed by compiler or assembler identification
.text,.data, .bss	C_STAT	DT_NON	T_NULL	section
tagname	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_NULL	tag name
.eos	C_EOS	DT_NON	T_NULL	end of structure
function name	C_EXT C_STAT	DT_FCN	Any except T_MOE.	function
array name	C_AUTO C_STAT C_MOS C_MOU C_TPDEF	DT_ARY	Any except T_MOE.	array
.bb	C_BLOCK	DT_NON	T_NULL	beginning of block
.eb	C_BLOCK	DT_NON	T_NULL	end of block
.bf,.ef	C_FCN	DT_NON	T_NULL	beginning and end of function
name related to structure, union, enumeration	C_AUTO C_STAT C_MOS C_MOU C_TPDEF	DT_PTR DT_ARR DT_NON	T_STRUCT T_UNION T_ENUM	name related to structure, union, enumeration

A tagname is a symbol name that includes the special symbol `.xfake`. The classes `fcname` and `arrname` represent any symbol name.

Filenames

Filenames can be any length. Filenames larger than 14 characters go into the string table. Shorter filenames are padded with zeros.

The `.file` symbol contains a 0 until the object file is linked. At the time, it either points to the file header for the next file in the chain or the start of the global storage area. Each `.file` entry generates one, two, or three auxiliary table entries. The first entry lists the name provided with the `.file` directive. If the program is a compiled source file, the second entry contains the `.ident` information from the compiler, such as the compiler's name and version with the present date and time. The third entry contains the assembler information, such as assembler identity and version.



NOTE. *All entry information is controlled by the environment variable I960IDENT. If the I960IDENT variable is not set, the assembler generates no `.ident` entries.*

If the source file is an assembly file, the second auxiliary table entry contains the assembler information and the symbol table contains no third entry.

Sections

The auxiliary table entries for a section have the format shown in Table C-21.

Table C-21 Format for Auxiliary Table Entries

Bytes	Declaration	Name	Description
0-3	long int	x_scnlen	section length
4-5	unsigned short	x_nreloc	number of relocation entries
6-7	unsigned short	x_nlinno	number of line numbers
8-23	-	-	unused (filled with zeros)

Tag Names

The auxiliary table entries for a tag name have the format shown in Table C-22.

Table C-22 Tag Name Entries

Bytes	Declaration	Name	Description
0-5	-	-	unused (filled with zeros)
6-7	unsigned short	x_size	size of structure, union, and enumeration
8-11	-	-	unused (filled with zeros)
12-15	long int	x_endndx	index of next entry beyond this structure, union, or enumeration
16-23	-	-	unused (filled with zeros)

End of Structure

The auxiliary table entries for the end of structure have the format shown in Table C-23.

Table C-23 Table Entries for End of Structure

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	-	-	unused (filled with zeros)
6-7	unsigned short	x_size	size of , union, or enumeration
8-23	-	-	unused (filled with zeros)

Functions

The auxiliary table entries for a function have the format shown in Table C-24.

Table C-24 Table Entries for Function

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-7	long int	x_fsize	size of function (in bytes)
8-11	long int	x_innoptr	file pointer to line number
12-15	long int	x_endndx	the end index for functions points to the symbol table entry for the next function, except the last function for the .file scope, which points at the first static symbol in the .file scope.
16-23	unsigned short	x_tvndx	unused (filled with zeros)

Arrays

The auxiliary table entries for an array have the format shown in Table C-25.

Table C-25 Table Entries for Array

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	unsigned short	x_inno	line number of declaration
6-7	unsigned short	x_size	size of array
8-9	unsigned short	x_dimen[0]	first dimension
10-11	unsigned short	x_dimen[1]	second dimension
12-13	unsigned short	x_dimen[2]	third dimension
14-15	unsigned short	x_dimen[3]	fourth dimension
16-23	-	-	unused (filled with zeros)

End of Blocks and Functions

The auxiliary table entries for the ends of blocks and functions have the format shown in Table C-26.

Table C-26 End of Block and Function Entries

Bytes	Declaration	Name	Description
0-3	-	-	unused (filled with zeros)
4-5	unsigned short	x_inno	number of lines in block
6-23	-	-	unused (filled with zeros)

Beginning of Blocks and Functions

The auxiliary table entries for the beginning of blocks and functions have the format shown in Table C-27.

Table C-27 Beginning of Block and Function Entries

Bytes	Declaration	Name	Description
0-3	-	-	unused (filled with zeros)
4-5	unsigned short	x_inno	line number in source where function begins
6-11	-	-	unused (filled with zeros)
12-15	long int	x_endndx	index of next entry past this block
16-17	-	-	unused (filled with zeros)

Names Related to Structures, Unions, and Enumerations

The auxiliary table entries for structure, union, and enumerations symbols have the format shown in Table C-28.

Table C-28 Entries for Structures, Unions, and Enumerations

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	-	-	unused (filled with zeros)
6-7	unsigned short	x_size	size of the structure, union, or numeration
8-17	-	-	unused (filled with zeros)

Auxiliary Entry Declaration

The C language structure declaration for an auxiliary symbol table entry is given in Example C-10. This declaration can be found in the header file `coff.h`.

Example C-10 Auxiliary Symbol Table Entry

```

/*
 *  AUXILIARY ENTRY FORMAT
 */
union auxent {
    struct {
        long    x_tagndx;    /* str, un, or enum tag indx */
        union {
            struct {
                unsigned short x_lnno; /* declaration line number */
                unsigned short x_size; /* str, union, array size */
            } x_lnsz;
            long    x_fsize; /* size of function */
        } x_misc;
        union {
            struct {          /* if ISFCN, tag, or .bb */
                long    x_lnnoptr; /* ptr to fcn line # */
                long    x_endndx; /* entry ndx past block end */
            } x_fcn;
            struct {          /* if ISARY, up to 4 dimen. */

```

continued 

Example C-10 Auxiliary Symbol Table Entry (continued)

```

unsigned short  x_dimen[DIMNUM];
    } x_ary;
} x_fcary;
unsigned short  x_tvndx; /* transfer vector index
                        (not used)*/

    } x_sym;
union {
char x_fname[AUXFILNMLEN]; /* File name for .file
                            symbol */

struct {
    long x_zeroes; /* zero indicating offset valid */
    long x_offset; /* symbol string table offset */
} x_n;
} x_file;
    struct {
long    x_scrlen; /* section length */
unsigned short  x_nreloc; /* number of relocation
                        entries */
unsigned short  x_nlinno; /* number of line numbers */
    } x_scn;
    struct {
long    x_tvfill; /* tv fill value */
unsigned short  x_tvlen; /* length of .tv */
unsigned short  x_tvran[2]; /* tv range */
    } x_tv; /* info about .tv section (in auxent of
            symbol .tv) */

/*
** i960 processor-specific *2nd* aux. entry formats
*/
    struct {
long    x_stindx; /* sys. table entry */
    } x_sc; /* system call entry */
    struct {
unsigned long   x_balntry; /* BAL entry point */
    } x_bal; /* BAL-callable function */

```

continued ➡

Example C-10 Auxiliary Symbol Table Entry (continued)

```
/*
** i960 processor 2nd and 3rd aux. entry formats
*/
struct {
    unsigned long    x_timestamp; /* identification time
                                stamp*/
    char    x_idstring[20]; /* producer identity string */
    } x_ident;
    char a[sizeof(struct syment)]; /* force aux to
                                syment size */
};

#define    AUXENT    union auxent
#define    AUXESZ    sizeof(AUXENT)
```

String Table

Symbol table names longer than eight characters are stored next to each other in the string table; each symbol name is delimited by a NULL byte. The first four bytes of the string table are the size of the string table in bytes; offsets in the string table are therefore 4 or more.

In this example, the file has two symbols whose names are longer than eight characters, `long_name_1` and `another_one`. Thus, the string table has the format shown in Figure C-3.

Figure C-3 String Table

Index	28			
0				
4	`l'	`o'	`n'	`g'
8	`-'	`n'	`a'	`m'
12	`e'	`-'	`1'	`\0'
16	`a'	`n'	`o'	`t'
20	`h'	`e'	`r'	`-'
24	`o'	`n'	`e'	`\0'

OSD321

The size of the string table in Figure C-3 is 28 bytes, which is equal to 4 bytes plus one byte for each character including the null terminator. The index of `long_name_1` in the string table is 4 and the index of `another_one` is 16.

Access Routines

Access routines, found in the common object file library, can be used for reading a common object file. These routines insulate the calling program from having to know the structure of the object file.

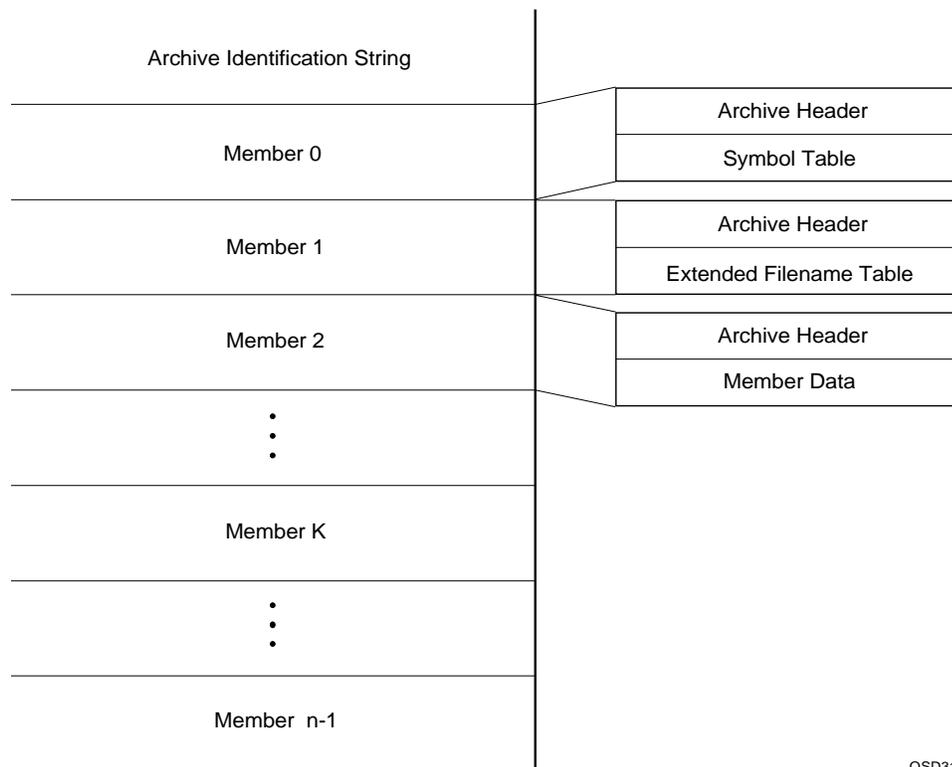
Archive Library Format

Figure C-4 represents a typical archive. The figure shows an object library consisting of n members.

The definitions of four elements constitute the common archive file format (CAFF):

- archive identification string
- one or more members
- symbol table
- name or spelling pool

The following sections describe each of these elements in greater detail.

Figure C-4 An Archive Library

OSD319

The Archive Identification String

The archive identification string identifies a file as an archive. Each archive library begins with a special string. For example, these two lines define the archive identification string:

```
#define ARMAG "!<arch>\n"
/*archive identification string*/
#define SARMAG 8
/*length of archive identification string*/
```

C

This string appears as the first eight characters in an archive. This string must be present, or the archiver cannot recognize this file as common archive file format (CAFF).

Archive Members

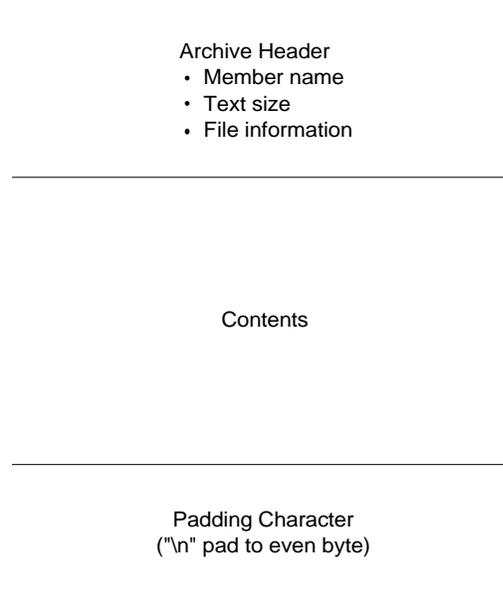
Archives can contain the following combinations of file types:

- COFF and ASCII text
- COFF only
- b.out format only

Archive members are arranged in sequential order within the library.

Figure C-5 represents a typical member, such as Member K of the archive shown in Figure C-4.

Figure C-5 An Archive Member



OSD320

The member header begins with the member name to identify the module within the archive. Several additional entries, containing control information, follow this string. A special trailer string terminates each archive member header.

The structure in Example C-11 defines the archive member header:

Example C-11 Archive Member Headers

```
#define ARFMAG "`\n"          /*header trailer string*/
struct ar_hdr               /*member header*/
{
    char ar_name[16];        /*-terminated member name*/
    char ar_date[12];        /*member date*/
    char ar_uid[6];          /*member user id*/
    char ar_gid[6];          /*member group id*/
    char ar_mode[8];         /*member mode(octal)*/
    char ar_size[10];        /*member size*/
    char ar_fmags[2];        /*header trailer string*/
};
```

Numeric information in the member header is stored in decimal format, except for `ar_mode`, which is formatted in octal. You can look at the information stored in the member headers of an archive by using the archiver's table-of-contents control with the verbose modifier on the command line.

Table C-29 lists archive member headers, their sizes, and their contents.

Table C-29 Size and Contents of Archive Member Headers

Bytes	Field	Contents
0-15	ar_name;	field contains the name of the member, padded with a slash (/) followed by blanks. The archiver derives this name from the pathname of the external file when it adds the member to the archive. The member name cannot be changed, although the member may be replaced, deleted, or moved within the archive.
16-27	ar_date	field shows the date and time of the external file when initially archived or updated in the archive. This date is returned from a system call; format of the date is system-dependent.
28-33	ar_uid	fields contain the user and group identification numbers of the user owning this member. On Windows hosts, these fields contain zero.
34-39	ar_gid	
40-47	ar_mode	field is derived from the system and contains an octal representation of the file permissions on the external file at archival time.
48-57	ar_size	field contains the size, in bytes, of the member. The member's size does not include the extra byte of padding, if present at the end of the archive member. Each archive file member starts on an even byte boundary, with a single new-line pad between members, if necessary. An archive member may not contain any empty areas.
58-59	ar_fmags	field contains the header trailer string ('\n').

When you add members with long names using the replace or update command, the archiver creates an extended filename table to store member names longer than 14 characters. If the archiver creates the extended filename table, the table follows the second archive header. If you strip the symbol table, the extended filename table follows the first archive member (see Figure C-4).

The Symbol Table

The first part of an archive (designated Member 0 in Figure C-4) is the archive symbol table. The archiver generates this structure when you add the first COFF or b.out format file to the archive. It is updated whenever necessary to reflect the current contents of the archive.

The symbol table is transparent to the user and inaccessible to a user of the archiver. It is implemented as a member of the archive, with a standard archive header. The symbol table has a name of zero length, that is:

```
ar_name[0]=='\0' ('\0' means NULL, the string terminator)
```

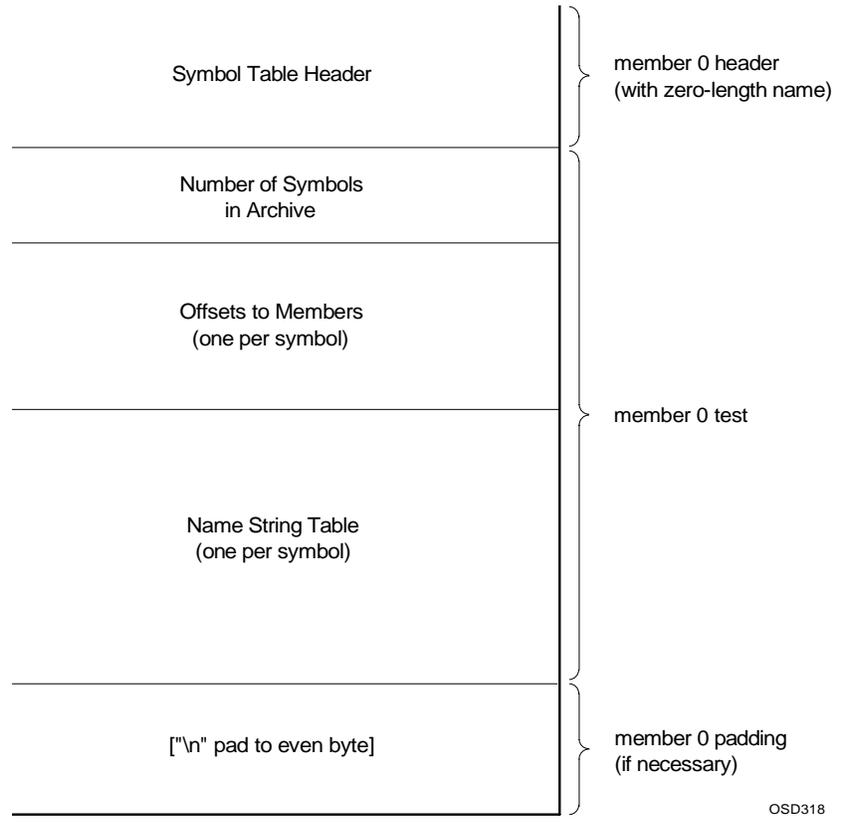
The `ar_date` field in the symbol table header reflects the date of the archive's creation. Figure C-6 illustrates an archive symbol table.

The symbol table consists of the following fields:

- Total number of symbols in the archive: 1 word.
- Array of offsets to member headers: 1 word per symbol.
- String table of null-terminated external symbols: 1 string per symbol.

The symbol table enables the linker to make a more efficient pass over object libraries.

Figure C-6 The Archive Symbol Table



OSD318

HP/MRI IEEE 695 Object File Format



This chapter describes the Microtec Research Inc./Hewlett-Packard Company version of IEEE-695 object module format, supporting assemblers, compilers, linkers and debuggers.

The material for this chapter is taken from the *HP/MRI IEEE-695 Object Module Format Specification*, Rev. 4.0, August 16, 1989. This chapter contains only those parts of the HP/MRI IEEE-695 specification that you need to understand output for the COFF to IEEE-695 converter (cvt960). This chapter omits the following:

- Target-specific information not pertinent to the i960® processor family.
- Translator output information for high-level languages other than C.
- Memory location information for relocatable modules. Only position-dependent (absolute) modules are described.
- Any other information not relevant to debugging and using output from the cvt960 utility.

If you need a complete IEEE-695 object module specification, you should consult the IEEE or MRI/HP version, as appropriate for your application. The Related Publications section in Chapter 1 provides ordering information for both.

The IEEE Trial Use 695 standard describes both an ASCII and a binary version of the format. MRI and HP implement the more compact binary form. Derived from the IEEE Trial Use Standard 695, the HP/MRI specification includes extensions and limitations required to support MRI and HP products.

D

Terminology

The IEEE specification defines a term that is redefined in this chapter. The term applies to the basic division of an object file that is referred to as a "command". Since this conflicts with the IEEE-695 use of command, the basic unit is renamed to be a "record". Object module records are predefined with a record type byte in the range \$E0 through \$FF. The term library is used throughout to mean a single file with more than one relocatable module. The term MAU is used throughout to mean minimum addressable unit; e.g. a byte (8 bits) on the i960® processor.

Table D-1 shows the initial bytes of IEEE formats described in this section.

Table D-1 Initial Bytes of IEEE Elements

Prefix	Description
\$00-\$7F	Simple number in the range 0 to 127, or 7-bit ASCII string with length 0 to 127.
\$80-\$84	Number larger than 127 or negative. 0 to 4 bytes follow. \$80 is used as a place holder and indicates that the value was not provided.
\$89-\$9F	Unused.
\$BE-\$BF	Function values.
\$C0-\$DA	Variable letters (null, A-Z).
\$DB-\$DD	Unused.
\$DE-\$DF	Extension length. If DE, the next byte is the length of an 8-bit string between 0 and 255 bytes long. If DF, the next two bytes in high-order/low-order format are the length of an 8-bit string between 0 and 65535 bytes long.
\$E0-\$FA	Record headers.
\$FB-\$FF	Unused.

Nomenclature

The following nomenclature is used throughout this chapter:

- Braces { } surround a required field.
- Brackets [] surround an optional field.
- Dollar Signs (\$) precede character representations of hexadecimal numeric values.

Number Format

Numbers are used to define byte counts for fields and to specify numeric parameters. These specifications can have two forms:

- If the value is between 0-127 decimal, the number is $\$0-\$7F$.
- If the value is greater than 127 decimal, then the number must be defined by 1 byte of count with the high order bit set ($\$80$) followed by the indicated number of bytes of numeric data with the most significant byte first. The range for the count is usually 0-4 (i.e. $\$80-\84) and can be 0-8 on some installations. This form is also valid for numbers in the range 0-127.

For example, $\$7FFF$ is encoded as $\{\$82\}\{\$7F\}\{\$FF\}$ (3 bytes).

0 can be encoded as $\{\$00\}$ or $\{\$81\}\{\$00\}$. 2^{32} can be encoded as $\{\$85\}\{01\}\{00\}\{00\}\{00\}\{00\}$, etc.

- Omitted optional fields in records may be represented by a byte count of zero. Example: $\{\$80\}$
- Numeric fields are represented in the chapter as $\{n\}$ and $\{x\}$.
- Numeric fields in miscellaneous records are represented as $\{v\}$.

Name Format

Name fields are represented in this chapter by $\{ID\}$ and consist of 1 byte of count (0-127) followed by the indicated number of ASCII characters. The HP/MRI format extends the IEEE specification to allow the use of any printable ASCII character in a name. Characters are represented as hexadecimal values in the file but are represented as quoted characters in this chapter for improved readability, as follows:

name "ABCD" = $\{\$04\}\{\$41\}\{\$42\}\{\$43\}\{\$44\}$

D

Name fields in miscellaneous records are represented as {s}.

The IEEE format allows only for printable strings. This implementation allows for non-printable strings.

An extension byte allows for more than 127 characters. If the reader encounters a DE character, the next one byte is the string length. The one byte length allows strings from 0 to 255 characters. If the reader encounters a DF character, the next two bytes are the string length. The two byte string allows 0 to 65535 characters.

Information Variables

Information variables convey information to a symbolic debugger or linker about various constructs within the program. The information conveyed relates to symbols, section addresses and lengths, starting addresses, and current PC value. These are represented by an alphabetic letter optionally followed by a number:

G Execution starting address.

I_n Address of public symbol *n*.

N_n Address of local symbol *n*.

P_n The program counter for section *n*; implicitly changes with each LR, LD, or LT that applies to section *n* in the Data part.

S_n The size, in minimum address units (MAUs), of a section *n*.

W_n *n* is 0 through 7; W_n is the file offset, in bytes, of the *n*'th part of the object file from the beginning of the file.

The number, if present for symbol definitions, identifies which of several variables of the same type is referenced. This number is referred to as an "index" in the discussion that follows. There are 3 different series of indices: external reference indices, section indices and public name/type/local name indices. Indices must be unique within a module for each series and must be included with all variable specifications except G. Public/local (I/N) type symbol indices between 0 and 31 are reserved for special class symbols. Normal symbol indices begin at 32. Therefore, "I3" represents public symbol number 3 in the current module.

Specification of G variables must not include an index. The IEEE standard has been extended to require index values for L, S, and P variables (these are all section indices). The binary encoding for the letters A-Z is \$C1-\$DA respectively.

Line Numbers

Object modules can have a significant number of line number records included in typical situations. To minimize the impact upon the size of the object module, the HP/MRI standard defines only one NN record per source file. A line number is specified by ATN and ASN records only.

Types

Symbol types supply information to debug and analysis tools to aid in determining the size, organization, and type of program object referenced by the symbol. Each symbol has an associated type number and/or a mnemonic "code letter" that serves as a shorthand identifier for the type in the object file and elsewhere.

Complex Types

Table D-2 identifies the supported high level complex types. These types must be explicitly defined using an IEEE 'TY' directive (see the Define Types (TY) section) in order to correctly represent the use of the symbol type in the high-level language source code. Table D-2 shows what parameters are used to define the type, where these parameters appear in the IEEE, and the NN and TY records which define the type.

Table D-2 HP/MRI IEEE-695 Object-file Representation of High-level Types

Definition	Type-parameters	IEEE Record/Field
Unknown type (sized)	'type name'	NN/{Id}
Mnemonic: !	! (\$21)	TY/{n3}
	size in MAUs	TY/{n4}
generalized C language	'enum-tag-name'	NN/{Id}

continued 

Table D-2 HP/MRI IEEE-695 Object-file Representation of High-level Types
(continued)

Definition	Type-parameters	IEEE Record/Field
enumeration	N (\$4E)	TY/{n3}
Mnemonic: N	0	TY/{n4}
	size of enumeration in MAUs	TY/{n5}
	1st enum constant name	TY/{n6}
	1st enum constant value	TY/{n7}
	additional names/values	[...]
32 bit pointer to another type	(name or null-name)	NN/{Id}
Mnemonic: P (pointer)	P (\$50)	TY/{n4}
	type index of pointer target	TY/{n4}
data structure	'structure-tag-name'	NN/{Id}
Mnemonic: S (structure)	S (\$53)	TY/{n3}
	size of structure (in MAUs)	TY/{n4}
	member 1 name	TY/{n5}
	member 1 type index	TY/{n6}
	member 1 MAU offset	TY/{n7}
	member 2 name	TY/{n8}
	member 2 type index	TY/{n9}
	member 2 MAU offset	TY/{n10}
	[additional members]	[...]
union of members	'union-tag-name'	NN/{Id}
Mnemonic: U (union)	U (\$55)	TY/{n3}
	size of union (in MAUs)	TY/{n4}
	member 1 name	TY/{n5}
	member 1 type index	TY/{n6}
	member 1 MAU offset	TY/{n7}
	member 2 name	TY/{n8}
	member 2 type index	TY/{n9}
	member 2 MAU offset	TY/{n10}
	[additional members]	[...]

continued 

Table D-2 HP/MRI IEEE-695 Object-file Representation of High-level Types
(continued)

Definition	Type-parameters	IEEE Record/Field
C array with lower bound = 0	(name or null-name)	NN/{Id}
Mnemonic: Z (zero based array)	A (\$5A)	TY/{n3}
	type index of component	TY/{n4}
	high bound (note 1)	TY/{n5}
Bitfield type	'type name'	NN/{Id}
Mnemonic: g	g {\$67}	TY/{n3}
	signed (0=unsigned,1=signed)	TY/{n4}
	size (in bits, 1 through n)	TY/{n5}
	base type index	TY/{n6}
procedure with compiler dependencies	'procedure-name'	NN/{Id}
Mnemonic: x (executable)	x (\$78)	TY/{n4}
	attribute (note 2)	TY/{n4}
	frame_type (note 4)	TY/{n5}
	push_mask (note 5)	TY/{n6}
	return_type	TY/{n7}
	# of arguments (note 3)	TY/{n8}
	[1st argument type]	TY/[n9]
	[2nd argument type]	TY/[n10]
	[additional argument types]	TY/[n11 thru nN]
	level (note 6)	TY/{n9 or TY/nN + 1}

NOTE 1: When the upper limits in array types A and Z are unknown, as for external references, the number of elements should be set to -1.

NOTE 2: The attribute parameter of the function type (X or x) is bit mask of:

Bit	Meaning	Bit	Meaning
0	Unknown frame - Set if this is a leaf procedure	5	PASCAL nested (always clear)
1	Near (always clear)	6	no push mask available (always set)
2	Far (always clear)	7	Interrupt (always clear)
3	Re-entrant (always set)	9-32	To be defined
4	ROMable (always set)		

NOTE 3: # of arguments (-1 if unknown).

NOTE 4: The 'frame-type' indicates the type of stable frame used by the routine. cvt960 sets the frame-type to 0.

NOTE 5: The 'push_mask' is always set to 0.

NOTE 6: the 'level' parameter is always set to 0.

D

Built-in Types

Table D-3 identifies the implicit or "built-in" types supported by the `cvt960` program. The built-in types represent C type definitions for common scalar types (and pointers to common scalar types) that are implicit to the compiler, assembler, linker, and debugger. As for complex types, the type number or mnemonic letter code for built-in types implies the size and organization of the program object. The type number also specifies a default type name for use by HP or MRI debugging tools in referring to the built-in type.

Built-in types normally do not require additional information other than the type number to completely describe them. Only the number of the built-in type is used in an ATN record describing a symbol having one of the implicit types. It is also the number used in the definitions for more complex types that have elements that are of built-in type. The shorthand notation for implicit types is intended to minimize the size of object modules by providing a short notation for the common subsets of more general types.

The interpretation of built-in types by `cvt960` is shown in Table D-3.

The following assumptions relating to typedefs are made by HP and MRI tools:

- Type "char" is assumed to be signed.
- The size assumed for a pointer is the natural size for the target (i.e. i960 = 4 minimum address units).

Table D-3 HP/MRI IEEE-695 Object-file Built-in Types

#	Mnemonic	Definition	Default Type Name
0	?	unknown type	'UNKNOWN TYPE'
1	V (void)	procedure returning void	'void'
2	B (byte)	8-bit signed	'signed char'
3	C (char)	8 bit unsigned	'unsigned char'
4	H (halfword)	16 bit signed	'signed short int'

continued ➡

Table D-3 HP/MRI IEEE-695 Object-file Built-in Types (continued)

#	Mnemonic	Definition	Default Type Name
5	I (int)	16 bit unsigned	'unsigned short int'
6	L (long)	32 bit signed	'signed long'
7	M	32 bit unsigned	'unsigned long'
10	F (float)	32 bit floating point	'float'
11	D (double)	64 bit floating point	'double'
12	K (king size)	extended precision floating point	'long double' (see note 1)
15	J (jump to)	code location	'instruction address'
26-31		reserved for future use	
32		pointer to unknown type	'UNKNOWN TYPE'
33		pointer to procedure returning void	'void'
34		pointer to 8 bit signed	'signed char'
35		pointer to 8 bit unsigned	'unsigned char'
36		pointer to 16 bit signed	'signed short int'
37		point to 16 bit unsigned	'unsigned short int'
38		point to 32 bit signed	'signed long'
39		pointer to 32 bit unsigned	'unsigned long'
42		pointer to 32 bit floating point	'float'
43		pointer to 64 bit floating point	'double'
44		pointer to extended precision floating point	'long double' (see note 1)
58-255		reserved for future use	

Object File Components

An object file is divided into seven component parts. Each part is a contiguous group of bytes within the file. The component parts may occur in any order within the file with the exception that the Header must occur first and the Module End must occur last. The Header part contains information pointing to the location of the other parts within the file. Therefore, the various file parts do not necessarily have to be read in the order in which they appear. The component parts listed below are described in the following sections:

Header Part

- Module Beginning (MB) - $\$E0$
- Address Descriptor (AD) - $\$EC$
- Assign Value to Variable W0 (ASW0) - $\$E2D700$
- Assign Value to Variable W1 (ASW1) - $\$E2D701$
- Assign Value to Variable W2 (ASW2) - $\$E2D702$
- Assign Value to Variable W3 (ASW3) - $\$E2D703$
- Assign Value to Variable W4 (ASW4) - $\$E2D704$
- Assign Value to Variable W5 (ASW5) - $\$E2D705$
- Assign Value to Variable W6 (ASW6) - $\$E2D706$
- Assign Value to Variable W7 (ASW7) - $\$E2D707$

AD Extension Part (ASW0)

- Variable Attributes (NN) - $\$F0$
- Variable Attributes (ATN) - $\$F1CE$
- Variable Values (ASN) - $\$E2CE$

Environment Part (ASW1)

- Variable Attributes (NN) - $\$F0$
- Variable Attributes (ATN) - $\$F1CE$
- Variable Values (ASN) - $\$E2CE$

Section Definition Part (ASW2)

- Section Type (ST) - $\$E6$
- Section Size (ASS) - $\$E2D3$
- Section Base Address (ASL) - $\$E2CC$

External Part (ASW3)

Public (External) Symbol (NI) - $\$E8$
 Variable Attribute (ATI) - $\$F1C9$
 Variable Values (ASI) - $\$E2C9$

Debug Information Definition Part (ASW4)

Declare Block Beginning (BB) - $\$F8$
 Declare Type Name, filename, line numbers, function name, variable names, etc.
 (NN) - $\$F0$
 Define Type Characteristics (TY) - $\$F2$
 Variable Attributes (ATN) - $\$F1CE$
 Variable Values (ASN) - $\$E2CE$
 Declare Block End (BE) - $\$F9$

Data Part (ASW5)

Current Section (SB) - $\$E5$
 Current Section PC (ASP) - $\$E2D0$
 Load Constant MAUs (LD) - $\$ED$
 Repeat Data (RE) - $\$F7$

Trailer Part (ASW6)

Execution Starting Address (ASG) - $\$E2C7$
 Module End (ME) - $\$E1$

Header Part

The header part contains information pointing to the location of other parts within the file.

Module Begin (MB)

The MB record must be the first record in the module.

```
{ $\$E0$ }{Id1}{Id2}
 $\$E0$     Record type
Id1     Processor (e.g. "80960CORE")
Id2     Module name
```

D

Table D-4 shows the i960® processor names that tools consuming HP/MRI IEEE-695 object files recognize in the `id1` field of the MB record.

Table D-4 Processor Names

Name	Processor Family
80960CORE	Intel i960 core architecture
80960KA	Intel i960 KA, SA
80960KB	Intel i960 KB, SB
80960CA	Intel i960 CA, CF
80960JX	Intel i960 JA, JD, JF, JL, RP
80960HX	Intel i960 HA, HD, HT

Address Descriptor (AD)

The AD record describes the characteristics of the target processor.

`{ $EC } { 8 } { 4 } { $CC }`

`$EC` Record type

8 Number of bits/minimum address unit

4 Number of minimum address units constituting the largest address form

`$CC` ('L') Low address of field contains least significant byte

Assign Value To Variable W0 (ASW0)

The ASW0 record contains a file byte offset pointer to the AD Extension record relative to the beginning of the file. A zero (0) value indicates that this extension is not included in the file.

`{ $E2 } { $D7 } { 00 } { n }`

`n` Byte offset in file in number format (see the AD Extension Part section)

Assign Value To Variable W1 (ASW1)

The ASW1 record contains a file byte offset pointer to the Environmental record relative to the beginning of the file. A zero (0) value indicates that this extension is not included in the file.

$$\{\$E2\}\{\$D7\}\{01\}\{n\}$$

n Byte offset in file in number format (see the AD Extension Part section)

Assign Value To Variable W2 (ASW2)

The ASW2 record contains a byte offset pointer to the module Section part relative to the beginning of the module. A zero (0) value indicates that this part is not included in the module.

$$\{\$E2\}\{\$D7\}\{\$02\}\{n\}$$

n Byte offset in file in number format (see the AD Extension Part section)

Assign Value To Variable (ASW3)

The ASW3 record contains a byte offset pointer to the module External part relative to the beginning of the module. A zero (0) value indicates that this part is not included in the module.

$$\{\$E2\}\{\$D7\}\{\$03\}\{n\}$$

n Byte offset in file in number format (see the AD Extension Part section)

Assign Value To Variable W4 (ASW4)

The ASW4 record contains a byte offset pointer to the module Debug Information definition part relative to the beginning of the module. A zero (0) value indicates that this part is not included in the module.

$$\{\$E2\}\{\$D7\}\{\$04\}\{n\}$$

n Byte offset in file in number format (see the AD Extension Part section)

D

Assign Value To Variable W5 (ASW5)

The ASW5 record contains a byte offset pointer to the module Data part relative to the beginning of the module. A zero (0) value indicates that this part is not included in the module.

{ $\$E2$ }{ $\$D7$ }{ $\$05$ }{ n }

n Byte offset in file in number format (see the AD Extension Part section)

Assign Value To Variable W6 (ASW6)

The ASW6 record contains a byte offset pointer to the module Trailer part relative to the beginning of the module. A zero (0) value indicates that this part is not included in the module.

{ $\$E2$ }{ $\$D7$ }{ $\$06$ }{ n }

n Byte offset in file in number format (see the AD Extension Part section)

Assign Value To Variable (ASW7)

The ASW7 record contains a byte offset pointer to the ME record relative to the beginning of the module.

{ $\$E2$ }{ $\$D7$ }{ $\$07$ }{ n }

n Byte offset in File in number format (see the AD Extension Part section)

AD Extension Part

The AD Extension Part contains information describing how the object module was created. This part is located after the header part and the AD record. It is pointed to by the W0 portion of ASW0. An NN record with a unique index associates ATN records defining the additional information. For more information on the syntax of records in the AD Extension Part, see the HP/MRI IEEE 695 Format Object File Semantics section. The AD Extension Part has the following format:

NN: { $\$F0$ }{ $n1$ }{Id}

ATN: { $\$F1$ }{ $\$CE$ }{ $n1$ }{ $n2$ }{ $n3$ }[$x1$][$x2$][Id]

D

Id	Symbol name
\$F1CE	ATN record type
n1	Symbol name index (must be same index as specified for its associated record)
n2	Symbol type index (0 = unspecified)
n3	Attribute definition: The attribute definitions for the Environmental Part appear in Table D-6 below.

Table D-6 Attribute Definitions for the Environmental Part

n3	Description
50	Creation date and time; requires one extra field [x1[x2[x3[x4[x5[x6]]]]]]]: x1 Year (e.g., 1990) x2 Month (1 - 12) x3 Day(1 - 31) x4 Hour(0 - 23) x5 Minute (0 - 59) x6 Second (0 - 59) The date and time are derived from the date and time of the COFF file. The year is encoded as a decimal number, not four hexadecimal digits. There is no ASN record.
51	Command line text; requires one extra field [Id] containing the command line. The command line is derived from the cvt960 command line. There is no ASN record.
52	Execution status; requires one extra field [x]: 0 Success There is no ASN record.
53	Host environment; requires one extra field [x1]: 4 HP-UX There is no ASN record.
54	Tool and version number used to create the module; requires three extra fields [x1], [x2], and [x3] defining the tool, version, and revision number. An optional fourth field [x4] is an ASCII character that defines the revision level (e.g. A, B, etc). The [x1] field contains 210, the cvt960 tool code. There is no ASN record.
55	Comments; requires one extra field [Id] specifying the comment string. There is no ASN record.

External Part

The External part contains records used to define global symbols from COFF. Variable miscellaneous records are also allowed in the External part. For more information on the syntax of records in the External and Public parts, see the HP/MRI IEEE 695 Format Object File Semantics section.

Public (External) Symbol (NI)

The Public Symbol provides for Public symbol definition and is optionally included in a module. Public symbol indices begin at 32. Indices 0 through 31 are reserved.

`{ $B8 } { n } { Id }`

`$B8` Record type

`n` Public index number, unique within an object file (must be > 31, 0 - 31 reserved)

`Id` Symbol name

Attribute Records (ATI)

`{ $F1 } { $C9 } { n1 } { n2 } { n3 } { n4 }`

`$F1C9` ATI record type

`n1` Symbol name index (this must be the same index as specified for the M record)

`n2` Symbol type index as follows:

- 0 Unspecified
- 2 Initialized data byte
- 5 Initialized data word
- 7 32 bit double word
- 10 32 bit floating point
- 11 64 bit floating point
- 12 10 or 12 byte floating point
- 15 Instruction address

`n3` Attribute definition: The attribute definitions are described in Table D-7.

D

n4 If n2 is non-zero, number of elements in the symbol type specified in n2

Table D-7 Attribute Definitions for the External Part

n3	Description
19	Static symbol generated by assembler. There is an ASI record specifying the address value.

Value Records (ASI)

The ASI record defines values for variables.

`{ $E2 } { $C9 } { n1 } { n2 }`

\$E2C9 Record type

n1 Symbol index (this must be the same index as specified for the NI record)

n2 value of symbol

Section Part

The Section part contains information defining the sections of the module. A "section" in this context is a contiguous area of memory. It may be absolute or relocatable, and may or may not have a name. All data minimum address units must be defined in a section.

For more information on the syntax of records in the Section Part, see the HP/MRI IEEE 695 Format Object File Semantics section.

Section Type (ST)

Each section must have exactly one section type record.

ASL and ASS records must appear after the ST record they refer to.

`$ { E6 } { n1 } { 1 } [Id] [n2] [n3] [n4]`

\$E6 Record Type

n1 Section index (index must be greater than zero and unique to this module)

- 1 Section type (only the new section types are described here)
- AS {\$C1}{\$D3} normal attributes for absolute sections.
Sections from different modules with these attributes,
whether they have the same name or not, are considered
to be unrelated.
- ASP {\$C1}{\$D3}{\$D0} absolute code
- ASD {\$C1}{\$D3}{\$C4} absolute data

Section Size (ASS)

The ASS record is required for all sections and defines the size for this section.

- {\$E2}{\$D3}{n1}{n2}
- \$E2D3 Record type
- n1 Section index An ST record must have occurred before this ASS record.
- n2 Section size (in minimum address units). This expression must be a simple number.

Section Base Address (ASL)

ASL records specify the section base address.

- {\$E2}{\$CC}{n1}{n2}
- \$E2CC Record type
- n1 Section index (this must be the same index as specified for the ATN record)
- n2 Section Base address (in minimum address units)

Debug Information Part

The Debug Information part contains records that define how to determine the symbol related information for a module at execution time. This is required for debuggers that provide high-level debugging capabilities.

For information on the syntax of records in the Debug Information Part, see the HP/MRI IEEE 695 Format Object File Semantics section.

Block Begin (BB)

The BB records are an extension to the IEEE-695 Trial Use standard. They provide definitions of debugging information related to the high level language definitions for types, scope and line numbers. They also provide assembly level language definitions for modules and local symbols. A block beginning with a BB is terminated with a BE record. BB records can be nested according to rules described below. Nested BB blocks can be used to capture scoping information. The types of BB blocks include:

BB1	Type definitions local to a module.
BB3	A module. A non-separable unit of code, usually the result of a single compilation, i.e. the symbols associated with a COFF <code>.file</code> symbol.
BB4	A global subprogram.
BB5	A source file line number block.
BB6	A local (static) subprogram.
BB10	An assembler debugging information block.
BB11	The module portion of a section.

The following list describes features of some of the blocks.

- BB1, BB3 and BB5 blocks usually occur together and in that order.
- BB1 blocks can be absent for modules that declare no local types.
- BB5 blocks immediately follow BB3 blocks in this implementation.
- A BB5 cannot occur without a BB3.
- Consecutive BB3 and BB5 blocks must refer to the same module.

Block Nesting. For a summary of block nesting rules, see Table D-8 below.

```

Module-Scope Type Definitions (BB1)
  NN and TY records
Module-Scope Type Definitions End (BE1)

High Level Module Block Begin (BB3)
  Global Variables (NN, ATN8, ASN)
  Module-Scope Variables (NN, ATN3, ASN)

  Module-Scope Function Block Begin (BB6)
    Local Variables (NN, ATN, ASN)
  Module-Scope Function Block End (BE6)
  Global Function Block Begin (BB4)
    Local Variables (NN, ATN, ASN)
    Local Function Block Begin (BB6)   High level
      Local Variables (NN, ATN, ASN)   Module Block
    Local Function Block End (BE6)     (one for each
  Global Function Block End (BE4)       high-level
High Level Module Block End (BE3)       module)
Source File Block Begin (BB5)
  NN,ASN,ATN, line numbers in source
Source File Block End (BE5)
Assembly Module Block Begin (BB10)
  Compiler Generated Global/External
  Variables (NH, ATK16, ASK)
  Compiler Generated Local Variables (NH,
  ATK16, ASK)

  Assembler Section Block Begin (BB11)
  Assembler Section Block End (BE11)
  Assembler Section Block Begin (BB11)
  Assembler Section Block End (BE11)
Assembly Module Block End (BE10)
Assembly Module Block Begin (BB10)
  Global/Extern Variables (KN,ATN19,ASN)
  Local Variables (KK, ATN19, ASN)
  Assembler Section Block Begin (BB11)   Assembly Level
  Assembler Section Block End (BE11)     Module Block
      (one for each
  Assembler Section Block Begin (BB11)   assembly level
  Assembler Section Block End (BE11)     module)
Assembly Module Block End (BE10)

```

D

Table D-8 below illustrates which of the blocks under Inner can be nested within the blocks listed under Outer. Some of the blocks require an outer block. For example, a BB4 block requires that its outer, enclosing block be a BB3. Similarly, a BB1 or BB2 block requires that its outer, enclosing block be the Debug Part, or debug.

Table D-8 Summary of Permitted Block Nesting

	Inner			Outer					
	BB1	BB2	BB3	BB4	BB5	BB6	BB10	BB11	debug
BB1	no	no	no	no	no	no	no	no	yes
BB2	no	no	no	no	no	no	no	no	*
BB3	no	no	*	*	no	*	no	no	yes
BB4	no	no	required	no	no	no	no	no	no
BB5	no	no	no	no	*	no	no	no	yes
BB6	no	no	yes	yes	no	*	no	no	no
BB10	no	no	no	no	no	no	*	no	yes
BB11	no	no	no	no	no	no	required	no	no

* Supported by HP/MRI-695 but not produced by cvt960.

The format for each block type is described below:

Block Type 1 - unique type definitions for module

```
{ $F8 } { $01 } { 0 } { Id }
```

\$F8 Record type

\$01 Block Type 1 - unique typedefs for module

0 Block size in bytes (0 = unknown)

Id Module name (from COFF .file symbol).

Block Type 3 - high level module scope beginning

```
{ $F8 } { $03 } { 0 } { Id }
```

\$F8 Record type

\$03 Block Type 3 - high level module scope beginning

0 Block size in bytes (0 = unknown)
 Id Module name (must be the same name as specified for BB1)

Block Type 4 - global function

{ \$F8 } { \$04 } { 0 } { Id } { 0 } { n3 } { n4 }

\$F8 Record type
 \$04 Block Type 4 - global function
 0 Block size in bytes (0 = unknown)
 Id Function name
 0 Number of bytes of stack space required for local variables
 (in minimum address units) (0 = unknown)
 n3 Type index for return value parameter and function information
 ('x' type), (0 = unknown)
 n4 The absolute address of the beginning of the code block.

Block Type 5 - filename for source line numbers

{ \$F8 } { \$05 } { 0 } { Id }

\$F8 Record type
 \$05 Block Type 5 - filename for source line numbers
 0 Block size in bytes (0 = unknown)
 Id Source filename

Block Type 6 - local function

{ \$F8 } { \$06 } { 0 } { Id } { n2 } { n3 } { n4 }

\$F8 Record type
 \$06 Block Type 6 - local function (static)
 0 Block size in bytes (0 = unknown)
 Id Function name
 n2 Number of bytes of stack space required for local variables (in
 minimum address units)
 n3 Type index for return value parameter and function information
 ('x' type) (0 = unspecified)
 n4 Offset (in minimum address units). The offset is the absolute
 address of the beginning of the code block.

n3 Section index
 n4 Offset (in minimum address units)

Optional fields may be null; but if any field is null and a later field is present, the omitted field must be filled with the `{§80}` construct. The relationship of blocks to variable attribute and variable value records (NN, ASN, ATN records) is preserved in the file. For variables that have an NN, ASN, ATN triple, these records must be together in the block structure definition (i.e., there can be no BB nor BE records between them). Block definitions may be nested.

Variable Names (NN)

These NN records declare variable names, type names and line numbers. The IEEE-695 Trial Use standard has been extended to allow duplicate local symbols to be defined, as long as the indices and the scoping are different.

This provides symbol definitions that are local to a specific section.

```
{§F0}{n}{Id}
§F0 Record type
n Name index number (must be > 31, 0-31 are reserved)
Id Name
```

Define Types (TY)

The TY record specifies that a type name represents an explicit type definition other than the implicit types predefined for use with HP/MRI language variables. Different types with the same name may be declared. This is supported by this specification by having multiple NN, TY pairs with the same name in the NN.

```
{§F2}{n1}{§CE}{n2}[n3][n4]...
```

§F2 Record type

n1 Type index unique within module (>255) (0-255 reserved for implicit types)

§CE Record type

D

n2	Local name index for symbol defined by NN record
n3, n4 . . .	Variable number of fields specifying additional type information as defined in Tables D-7 and D-8.

Attribute Records (ATN)

Each ATN record is associated with an NN record and defines a valid symbol.

NN record:	{ \$F0 } { n1 } { Id }
ATN record:	{ \$F1 } { \$CE } { n1 } { n2 } { n3 } [x1] [x2] [x3] [x4] [x5] [x6] [Id]
ASN record:	{ \$E2 } { \$CE } { n1 } { h2 }
\$F0	NN record type
n1	Symbol name (NN record) type
Id	Symbol name
\$F1CE	ATN record type
n1	Symbol name index (this must be the same index as specified for the NN record)
n1	Symbol type index (0=untyped)
n1	The numbers representing the attributes, the blocks they can appear in, and their descriptions are illustrated in Table D-9.
x1 Id	Optional features, described for each attribute.
\$E2CE	ASN record type
n1	Symbol name (NN record) index
n2	Symbol value

Table D-9 Attribute Numbers, Blocks, and Descriptions

n3	Block	Description
1	4,6	Automatic variable; requires an additional field [x1] defining the stack offset (in minimum address units) . There is no ASN record.
3	3,4,6	Compiler defined static variable. There must be an ASN or ASI record specifying the address value.
5		External variable definition. There is no ASN record.
7	5	Line number; requires two extra fields giving the line number and column number. Two optional fields [x3] and [x4] are reserved and should be omitted. The line and column number represent the end of a group of one or more lines in a statement. A column number of 0 represents the end of the line and reflects the fact that cvt960 cannot get this information from COFF. Line numbers do not have to be in ascending order, and it is the consuming tool's responsibility to handle numbers that are "out of order." There must be an ASN record specifying the address.
8	3	Compiler global variable. There must be an ASN record specifying the address value.
10	4,6	Defines a variable name as a locked register; requires an extra field, [x1], to define the index of the register name. There is no ASN record.
19	10	Static variable generated by assembler; may be global in scope. There must be an ASN record specifying the address/value. There is one required field [x1], which indicates the number of elements of type n2 described by the symbol, and [x2], which is a local/global indicator. [x2]=omitted or 0 indicates local. [x2]=1 indicates global.
37, 38, 39, 50, 51, 52, 53, 54, 55		See the AD Extension Part section and the Environmental Part section.

continued ➡

D

Table D-9 Attribute Numbers, Blocks, and Descriptions (continued)

n3	Block	Description
62	4,6	Procedure block misc.; followed by two fields that describe the most recent procedure block. The first field [x1] is the pmisc. type identification number, the second [x2] is the number of additional ATN 65 or ASN records associated with this directive. See the Miscellaneous Records section for the codes associated with this directive.
63	3,4,6	Variable misc.; followed by two fields that describe a variable. The first field [x1] is the vmisc. type identification number, the second [x2] is the number of additional ATN 65 or ASN records associated with this directive. See the Miscellaneous Records section for the codes associated with this directive.
64	3	Module misc.; followed by two fields that describe the current module block. The first field [x1] is the mmisc. type identification number, the second [x2] is the number of additional ATN 65 and ASN records associated with this directive. See the Miscellaneous Records section for the codes associated with this directive.
65	3,4,6	Misc. string; requires one field that is a string value for miscellaneous records 62, 63 and 64.

Value Records (ASN)

The ASN records are used to define values for variables.

```
{ $E2 } { $CE } { n1 } { n2 }
```

\$E2CE Record type

n1 Symbol name index (must be the same as specified for the record)

n2 value for symbol (in minimum address units if it is an address)

Stack relative symbols and register-based symbols must not have an ASN record since the value is defined at execution time.

D

Block End (BE)

The BE record extends the IEEE standard and is used in conjunction with a BB record. The BE record for type 4,6, and 11 BB records are different than others as indicated in the following definitions:

Block End - General

{*\$F9*}

\$F9 Record type

Block End - for block types 4 and 6

{*\$F9*}{*n1*}

\$F9 Record type

n1 Expression defining the ending address of the function (in minimum address units)

Block End - for block type 11

{*\$F9*}{*n1*}

\$F9 Record type

n1 Expression defining the size in minimum address units of the module section

Data Part

The data part contains records defining fixed data for the module and is always loaded at the current PC value in the current section. The current section is defined by the SB record and the PC is defined by the ASP record. If no SB record is defined, the current section is specified as 0. If no ASP record is defined, the PC for a section is initially set to the start of the section.



NOTE. *Section 10.1 of the IEEE Trial Use Standard says that the current section is 0 before any SB records are encountered. Section 10.2 specifies that if no ST record is present for a section, the type is absolute and shall have an assignment to its L variable. Taken together, these statements imply that the example module in Section 4.1 of the standard is illegal. HP and MRI follow the definition as stated in Section 10.1 of the IEEE Trial Use Standard.*

Set Current Section (SB)

The SB record defines the current section. SB has no effect on the P variable.

```
{ $E5 } { n1 }
$E5      Record type
n1       Section index
```

Set Current PC (ASP)

The ASP record sets a new value for the current PC. An ASP record is required after an SB record to reset the value of the P variable.

```
{ $E2 } { $D0 } { n1 } { n2 }
$E2D0   Record type
n1       Section index
n2       Expression defining new value (in minimum address units)
```

Load Constant Bytes (LD)

The LD record specifies the number of minimum address units to be loaded as constant data.

```
{ $ED } { n1 } { . . . }
$ED     Record type
n1      Number of minimum address units (1-127)
. . .   (n1 x minimum address unit size) data bytes
```

D

Repeat Data (RE)

The RE record specifies data initialization in a compact form.

```
{ $F7 } { n1 }
```

\$F7 Record type

n1 Expression defining number of times to repeat the following LD or LR record data. The IEEE-695 standard has been extended to include repeating LD records. The length of data that can be repeated is limited to 128 bytes.

Trailer Part

The Trailer part contains the records described below.

Starting Address (ASG)

The ASG record is optional and defines the execution starting address. This expression requires \$BE/\$BF delimiters.

```
{ $BE } { n1 } { $BF }
```

\$E2C7 Record type

n1 Value defining the execution starting address (in minimum address units)

Module End (ME)

The ME record defines the end of the module and must be the last record in the module.

```
{ ME }
```

HP/MPI IEEE-695 Format Object File Semantics

This section describes the HP/MRI IEEE-695 format object file semantics. The format shows the records by record header (for example, NN is a name index record). Records enclosed in square brackets ("[" and "]") are optional; records enclosed in curly braces ("{" and "}") are repeatable 0 or more times.

AD Extension Part and Environment Part

The AD Extension part and the Environment part constitute attribute records describing the file, its contents, and its creation. The format of the two sections is shown below:

```
{ [NN]ATN[ASN] }
```

where at least one NN must be present before any ATN, and the name index for the ATN must be the same as the last NN.

Public/External Part

The Public/External part contains records describing public and external symbols, by name, type, and address. The format of the records is shown below:

```
Public: { NI [ATI ASI] }
```

where the name index for the NI, ATI and ASI records must match in each triplet. It is not possible to have more than one ATI or ASI record for any name. A vmisc may follow any public.

Section Part

The Section part describes the different sections in the file. It describes the combined sections after linkage.

The format of these records is shown below:

```
{ ST [ASS][ASL] }
```

where the section index for the ST, ASS, or ASL records must match for each group. It is not possible to have more than one ASS or ASL record for any section name.

D

Debug Part

In the Debug part, there are two types of main groups: high-level blocks created by a compiler, and assembly language blocks created by the assembler. The high level blocks contain all compiler symbol information, as described in the HP/MRI IEEE-695 specification. The format for the Debug part is shown below:

```
{ ([BB1] BB3 [BB5] [BB10]) or BB10 }
```

where the first enclosing parenthesis shows a high-level group: The module names for BB1, BB3, and BB10 must match; the filename in the BB5 is related to the module name. The BB10 block provides backward compatibility. The lone BB10 block is the assembly level group. It is created when there is no high-level information.

BB1 Block

A BB1 block contains type information for high-level symbols; it is described earlier in this document. The block is formatted as shown below:

```
{NN {TY}}
```

where any number of types with the same name is allowed. The name index must match between the TY record and the last NN record.

BB3 Block

A BB3 block contains the symbolic information for all symbols except types and lines. It represents one compilation unit (a full compilation module, with include files). It is formatted as shown below:

```
{[BB4] [BB6] NN ATN[ASN]}
```

where BB4 blocks are global functions and BB6 blocks are static functions or unnamed blocks. The NN, ATN, ASN pairings are public, static, or external symbols (locals are in BB4 and BB6 blocks). The name index for NN, ATN and ASN records must match.

BB4 and BB6 Blocks

BB4 and BB6 are scoping blocks and represent functions (procedures). They contain all local symbols to the function. BB6 blocks may nest inside of BB4 and BB6 blocks. If the BB6 block has a null name, it is a scoping block only ("{" blocks in C). The BB4 and BB6 blocks are formatted as shown below:

```
{ [BB6] ([NN] ATN) or (NN ATN[ASN]) }
```

where at least one NN record must be present for each ATN and ASN name index used. The optionality of NN records is available only for special ATN records (register lifetime). A local variable with the same name as another symbol in an outer block must still have a new NN record. The NN, ATN, or ASN records that describe a symbol must all reside within the same BB/BE scope; their affiliation cannot cross BB or BE boundaries. The optionality of ASN records is defined earlier in this document.

BB5 and BB10 Blocks

A BB5 block carries the source file information, such as the source filename, include filenames, and lines. It is formatted as shown below:

```
{ [BB5] [[NN] ATN ASN] } [BB10]
```

The BB10 block is created by the assembler to hold assembly language information, such as assembly language source filename, and local section information (R_Label sections); it is not intended to have assembler symbols such as a lone BB10 block.

The cvt960-created `.global_non_init` module, however, has many assembler-level globals in B10.

The BB10 block is placed after the BB5 block. The HP/MRI specification allows other BB5 blocks to be nested inside, interspersed in the line information. These are usually `include` files. The cvt960 converter does not produce nested BB5s because `include` file information is not available from COFF.

D

The line information must have at least one NN record before any ATN or ASN records, and the index for the ATN and ASN must be the same as the last NN. Only lines that have code associated with them need to be present. All readers can assume that any missing lines are associated with the next line specified.

The column offset parameter indicates the position of the high level source line. The offset is taken to mean the column position of the last token of the source text associated with the machine instructions immediately following the code position indicated by ATN 7.

Source file columns are numbered starting with 1. The special column offset value of 0 is defined to indicate the position of the last column on the line. Because column information is not available from COFF, the column is 0.

The BB10 block carries the information derived by the assembler from a file not produced by a compiler (or one that did not put in debug information). The BB10 block has the section information on a module basis (as opposed to the linker's combined sections); this allows a tool to know the part of a section that came from a particular module. Also, any local or global assembly language symbols are shown here. It is formatted as shown below:

```
{BB11 }{[BB10] NN ATN ASN}
```

where the BB11 block contains the section information as described earlier in this manual. BB10 blocks may be nested for `include` files. The name index for the NN, ATN and ASN records must match.



NOTE. *COFF lacks module-membership information for global uninitialized variables, so `cvt960` produces the pseudo-module `.global_non_init` for them.*

Miscellaneous Records

Miscellaneous records provide a flexible and extensible method for communicating information generated by a compiler or other language translator directly to a debugger or other consumer tool via the object file. Information in miscellaneous records is classified according to a coding system defined below. The content and meaning of each miscellaneous information category can be defined to suit a wide range of information needs, and new miscellaneous information categories can be defined as needed. Thus, miscellaneous records allow the IEEE-695 object module format to evolve in an orderly manner as new debugging features and requirements emerge.

One of the main advantages of miscellaneous records is that, in general, they are processed in a generic, content-independent manner by intermediate language system tools such as assemblers and linkers. That is, assemblers and linkers need not interpret or manipulate in any special way the contents of miscellaneous records, except to resolve, in the standard manner, the values of relocatable expressions that may be present in these records. As a result, there is no need to modify assemblers or linkers when new classes of miscellaneous information are defined.

Three classes of miscellaneous records have been defined: module miscellaneous records for augmenting the debugging information for program modules, procedure miscellaneous records for decorating code blocks, and variable miscellaneous records for decorating data objects. The three kinds of miscellaneous records differ primarily with regard to the scope within which the record's information applies. The affiliation of a miscellaneous record with the object or objects it describes is determined primarily by the relative position within the object file of the miscellaneous record and the object or objects it describes. These positional relationships are explained in more detail below.

Module Miscellaneous Records

Module miscellaneous records convey information about a program module. For high level modules, module miscellaneous records appear within a BB3/BE3 scope. For assembly modules, module miscellaneous records appear within a BB10/BE10 scope. The information in a module miscellaneous record applies to the module within whose scope the record is enclosed. For example, the information in the module miscellaneous record having code 50 (compiler Id, type checking rules, and compilation time) applies to the entire module and all objects in the module.

Multiple miscellaneous records can coexist within the same module scope, especially if the records have different classification codes. However, some module miscellaneous record types (e.g., code 50) allow at most one record of a given classification code within any single program module scope.

Specific object file readers may impose further restrictions on the position of module miscellaneous records. For example, if the information in a module miscellaneous record influences the interpretation of the debugging information of other objects in the module scope, specific consuming tools may require that the module miscellaneous record occur before any other debug information. This is strictly a requirement of the consuming reader tool, however, and not the object module format.

Variable Miscellaneous Records

The information in a variable miscellaneous record applies to the most recent data object declared using a normal NN/ATN/ASN cluster, as described in Chapter 3. For example, variable miscellaneous code defines the register shadowing parameters for the specific data object immediately preceding the variable miscellaneous record.

Procedure Miscellaneous Records

The information in a procedure miscellaneous record applies to the entire code block within whose scope the record is enclosed. The traditional scope for a procedure miscellaneous record has been a procedure or function code block, that is, BB4/BE4 or BB6/BE6. For example, procedure miscellaneous code 1 conveys the address of the exit (return) instruction of a procedure.

However, in anticipation of supporting future lexical features such as Ada tasks and package scopes, and to limit the proliferation of terminology associated with decoration of code blocks, the definition of the enclosing scope for a procedure miscellaneous record is broadened to include other kinds of code blocks, some of which are yet to be defined. There is no ambiguity regarding the scope of each procedure miscellaneous record, because the relevant enclosing scope had to have been agreed to both by the producer and consumer of the procedure miscellaneous information when the classification number was assigned.

Thus, by definition, each procedure miscellaneous record's classification number also implies the record's relevant code block scope.

General Syntax Rules

Miscellaneous records are composed of groups of NN, ASN, and ATN records that together form a cluster or packet of information. The miscellaneous record cluster can be thought of as a list of parameters, the first of which constitutes a classification number that dishes each cluster from all others. Remaining parameters are the information conveyed by the record cluster.

Parameters In Miscellaneous Records

Parameters in miscellaneous records may be character strings, numerical constants, compiler labels, or relocatable expressions. The object module format constrains neither the number of parameters a miscellaneous record may have, nor which of the permissible parameter types is to be used in any of the individual parameter slots, except the first slot. The first slot

must be the numerical information classification code. The number and composition of the remaining parameter slots is completely determined by the syntax specification for each miscellaneous record. However, the following rules apply to individual parameter values:

1. Numerical constants may be signed and have absolute values between 0 and $2^{31} - 1$.
2. Floating point constants must be represented as quoted strings.

Every parameter that is a number or relocatable expression is represented in the miscellaneous record cluster by an individual ASN record and every parameter that is a string is represented by an individual ATN.

Examples

The following example illustrates how a module miscellaneous record having the classification code 99 (chosen for illustration) would be documented in this specification, and how it would be encoded in the IEEE-695 object module.

```
code 99, value1, value2, 'string1', value3, 'string2'
```

The miscellaneous record cluster would be represented in this document as follows:

```
NN:      {$F0}{index}{null_name}
ATN:     {$F1}{$CE}{index}{$00}{$40}{$63}{5}
ASN:     {$E2}{$CE}{index}{value1}
ASN:     {$E2}{$CE}{index}{value2}
ATN:     {$F1}{$CE}{index}{$00}{$41}{string1}
ASN:     {$E2}{$CE}{index}{value3}
ATN:     {$F1}{$CE}{index){$00}{$41}{string2}
```

The parameters of the NN and first few ATN and ASN records have the following meanings:

NN: { $\$F0$ }{index}{null-name}
 index unique index within the current BB3 block
 null-name \$00 (i.e., null name string)

ATN: { $\$F1$ }{ $\$CE$ }{index}{n2}{n3}{n4}{n5}
 index unique index within the current BB3 block (the ATN index must
 match the index of the most recent NN record)
 n2 \$00 symbol type undefined
 n3 \$40 ATN type 64 - module miscellaneous information (mmisc)
 record
 n4 \$63 = module miscellaneous information code 99 ($\$63$ hex = 99
 decimal)
 n5 \$05 = number of additional ASN/ATN records (5) associated
 with this mmisc cluster

ASN: { $\$E2$ }{ $\$CE$ }{index}{n2}
 index as above
 n2 expression for value1 (etc.)

A special case is where the first parameter in the miscellaneous record cluster (after the classification code) is a string. In this case, the initial string is encoded in the first ATN of the miscellaneous record cluster right after the parameter indicating the number of additional ATN/ASN records in the cluster. A procedure miscellaneous record cluster matching this description, with illustrative classification code 99, is illustrated below.

code 99, 'string1', value1

The miscellaneous record would be represented in this specification as follows:

NN: { $\$F0$ }{index}{null_name}
 ATN: { $\$F1$ }{ $\$CE$ }{index}{ $\$00$ }{ $\$3E$ }{ $\$63$ }{1}{string1}
 ASN: { $\$E2$ }{ $\$CE$ }{index}{value1}



NOTE. *string2* is included in the first ATN record of the cluster.

Optional Parameter Fields

Some miscellaneous records have optional parameters. These are denoted in the parameter list as [*parameter*]. If some optional parameters in a record are present but others are not, all of the optional parameter slots preceding a supplied optional parameter must be accounted for. Missing optional parameter(s) whose values are numbers are indicated using the IEEE-695 "unknown" code (§80) in the slot corresponding to the missing parameter. If the missing optional parameter is a string, a §80 length string would appear in the ATN record corresponding to the missing string. Omitted optional parameters that follow the last supplied parameter need not be explicitly included in the miscellaneous record.

For example, in the following variable miscellaneous record, optional `value2` is missing:

```
code 99, value1 [,value2] [,value3]
```

The miscellaneous record cluster would be represented in this document as follows:

```
NN:      {$F0}{index}{null_name}
ATN:     {$F1}{$CE}{index}{$00}{$3F}{$63}{5}
ASN:     {$E2}{$CE}{indexXvalue1}
ASN:     {$E2}{$CE}{index}{$80}
ASN:     {$E2}{$CE}{index}{value3}
```

As another special case, it is permissible to omit the NN record when a variable miscellaneous directive immediately follows the variable NN/ATN/ASN that it modifies. In this case, the variable miscellaneous directive ATNs and ASNs would all use the same NN index as the actual variable:

```

NN index
ATN index info...           Original Variable
ASN index expression
ATN index 0 64 misc_code count
                               Variable Misc. Information
ASN/ATN index...
```

Codes for Miscellaneous Records

Each module, procedure, and variable miscellaneous directive is assigned an ID number from a common index pool. For example, there is only one miscellaneous directive with ID code 1.

The first 50 codes (0-49) are reserved for miscellaneous directives in which the assembler needs to correlate the argument information with other debug information. Codes greater than 49 are used for miscellaneous directives where the assembler only needs to encode the parameters of the miscellaneous directive into the relocatable object module.

Policies for Adding and Modifying Miscellaneous Records

Adding new miscellaneous records to the object module format is straightforward, but requires agreement between the producers and consumers of the miscellaneous information. To supplement the debugging information for some program object, the compiler designer need only agree with the consumer tool designer on the miscellaneous record classification code to identify the new information category, and output the appropriate miscellaneous record and parameters using the general rules described below.

Clearly, even when there is agreement between producers and consumers of IEEE-695 object modules, wholesale modification of existing miscellaneous records is undesirable, because older versions of object module reader tools can become confused by the new record syntax, and the backward compatibility of new object files with older consumer tools could suffer.

Policies for Generating and Reading Miscellaneous Records

The following policies are set forth to ensure that miscellaneous information records are created, modified, and consumed in an orderly manner:

1. (For object module producers) Adding new required fields to an existing miscellaneous record is prohibited. Adding new optional fields to an existing miscellaneous record is permissible only if (a) the new information is related to the information already in the record, and (b) the new information does not alter in any way the interpretation of the information already in the record. If the new information violates any of these conditions, the new information should go into a new miscellaneous record classification.
2. (For object module consumers) When a miscellaneous record having an unknown classification is encountered, the object file reader should first consult the version number of the object module format in the ATN code 37 record of the Environment Part (see Chapter 3). If the object module version is newer than the reader was designed to consume, the reader should provide an indication to the user to this effect. Readers might then either (a) continue reading the object file, simply ignoring the information, or (b) abort reading the object file with a message that continuing would result in potentially ignoring important information about symbols in the object module. If behavior (b) is implemented, the reader might be designed so that the user can force it to read the object file anyway, thereby allowing the user to get some benefit from the object file, while being fully aware of the consequences of ignoring some information.

When a miscellaneous record having an unexpected optional parameter is encountered, the situation is somewhat different. Assuming the unexpected optional parameters conform to the producer rules above, that is, they do not alter the interpretation of the information already in the record, they are truly discretionary, and the reader probably can safely proceed with reading the object module, perhaps after printing a warning (after checking the object module version number) to the effect that unexpected fields were encountered while reading a miscellaneous record.

Lastly, if the version number of the object file is one the reader was designed to consume, and either new miscellaneous codes or unexpected fields are encountered, the object module is either defective or a producer has broken the rules for that object module version. In either case, the reader should treat the remainder of the object file with the same (or perhaps greater) suspicion than when the object module version is newer than that supported by the reader.

The behavior in the face of all these contingencies is, of course, left to the implementor of the consumer tool.

The currently defined miscellaneous information records are documented in Table D-10.

D

Table D-10 Miscellaneous Record Codes

Misc. Type	Code	Meaning
module	50	Compiler Id and date stamp. Syntax: code 50, tool_code, type_rule, pointer_size [, 'compiler_version_string'] [, date] NN: {\$F0}{index}{null_name} ATN: {\$F1}{\$CE}{index}{\$00}{\$40}{\$32}{#_of_ATN/ASNs - currently between 3 and 10 inclusive} ASN: {\$E2}{\$CE}{index}{tool_code} ASN: {\$E2}{\$CE}{index}{type_rule} ASN: {\$E2}{\$CE}{index}{pointer_size} ATN: {\$F1}{\$CE}{index}{\$00}{\$41} {compiler_version_string} ASN: {\$E2}{\$CE}{index}{year} ASN: {\$E2}{\$CE}{index}{month} ASN: {\$E2}{\$CE}{index}{day} ASN: {\$E2}{\$CE}{index}{hour} ASN: {\$E2}{\$CE}{index}{minute} ASN: {\$E2}{\$CE}{index}{second}

continued ➡

Table D-10 Miscellaneous Record Codes (continued)

Misc. Type	Code	Meaning
variable procedure	63	<p>Call optimization information (i960-)</p> <p>Purpose/meaning: Holds the .sysproc/.leafproc information for the most recent public or external.</p> <p>Position: In the Public/External Part after NI/ATI/ASI triples or after NX records in relocatable files.</p> <p>Syntax: Code 63, proc_type, system_index bal_address</p> <p>NN: {\$F0}{index}{null_name}</p> <p>ATN: {\$F1}{\$CE}{index}{\$00}{\$3F}{\$3F}{#_of_ATN/ASN's (1 or 2)}</p> <p>ASN: {\$E2}{\$CE}{index}{proc_type}</p> <p>ASN: {\$E2}{\$CE}{index}{bal_address}</p> <p>Parameter meanings:</p> <p>proc_type: 0 if unknown 1 if leaf procedure 2 if system table procedure</p> <p>system_index: index into the system_table (used only if proc_type == 2)</p> <p>bal_address: branch and link address (used only if proc_type == 1)</p>

For more information on compiler identification, see the `.ident` directive in the *i960® Processor Assembler User's Guide*.

Index

A

a (After) archiver option modifier, 2-7
A (architecture) linker option, 7-25
arc960 command, 2-1
archiver, 2-1

- a (After) option modifier, 2-7
- b (Before) option modifier, 2-8
- c (create) option modifier, 2-9
- d (Delete) option modifier, 2-10
- F (Library Format) option modifier, 2-11
- h (Help) option modifier, 2-11
- i (Insert) option modifier, 2-8
- l (Local) option modifier, 2-12
- m (Move) option modifier, 2-13
- o (Output Date) option modifier, 2-14
- options summary, 2-2
- p (Print) option modifier, 2-14
- r (Replace) option modifier, 2-15
- s (Symbol Table) option modifier, 2-16
- t (Table of Contents) option modifier, 2-17
- u (Update) option modifier, 2-18
- v (Verbose) option modifier, 2-19
- V (Version) option modifier, 2-20
- v960 (Version) option modifier, 2-20
- x (Extract) option modifier, 2-21
- z (Suppress Time Stamp) option modifier, 2-22

B

b (Before) archiver option modifier, 2-8
B (section start address) linker option, 7-27
b.out / COFF / ELF converter, 3-1

- options summary, 3-2

backslash (\) character, use with tools, 1-5
BSS sections, linker, 7-4

C

c (circular library search) linker option, 7-29
c (Create) archiver option modifier, 2-9
C (startup alternative) linker option, 7-29
CAFF format for i960® processors, C-1
callj/calljx, link time optimization, 7-16
checksum rom960 directive, 12-7
cof960 command, 3-2

- object converter, 3-1

COFF / b.out / ELF converter, 3-1

- options summary, 3-2

COFF format for i960® processors, C-1
COFF symbol translation, cvt960, 4-5
COFF to IEEE-695 converter, 4-1

- options summary, 4-2

Common Object File Library (COFL), B-1
compatibility of mpp960, 8-1
conversion tools for object files, 3-1, 12-1
converter, 3-2

- output file, 3-3
- coverage analyzer, 5-1
 - controls summary, 5-2, 5-3
 - option summary, 5-3, 5-4
- CTOOLS 5.1 compatibility, 1-3
- cvt960 invocation, 4-1
- cvt960 object converter, 4-1
 - archives and relocatable objects, 4-3
 - COFF line numbers, 4-4
 - compilation/assembly information, 4-4
 - global uninitialized symbols, 4-3
 - IEEE-695 built-in types, 4-5
 - IEEE-695 converter warning messages, 4-8
 - Position-independent code, data, and symbols, 4-3
 - unreferenced types, 4-3
- D**

 - d (define common symbol space) linker option, 7-31
 - d (Delete) archiver option, 2-10
 - D (inhibit CAVE section compression) linker option, 7-31
 - debugging macros, 8-16–8-19
 - defsym (define a symbol) linker option, 7-32
 - Displaying archive structure, 6-12
 - dmp960
 - archive support, 6-11
 - displaying archive structure, 6-12
 - dumping absolute symbols, 6-5
 - dumping archive member contents, 6-14
 - examples, 6-6
 - section headers, 6-11
 - symbol tables, 6-9
 - with symbolic disassembly, 6-8
 - dmp960 / gdm960
 - options summary, 6-2 thru 6-4
 - dmp960 invocation, 6-1
 - dumper
 - archive support, 6-11
 - displaying archive structure, 6-12
 - options summary, 6-2, 6-3
 - dumper / disassembler, 6-1
 - dumping absolute symbols, 6-5
 - dumping archive member contents, 6-14
- E**

 - e (entry point) linker option, 7-33
 - ELF / b.out / COFF converter, 3-1
 - options summary, 3-2
 - ELF/DWARF sections, linker, 7-4
 - environment variables, linker, 7-20
- F**

 - f (fill) linker option, 7-35
 - F (format) linker option, 7-34
 - F (Library Format) archiver option, 2-11
- G**

 - G (big-endian target) linker option, 7-36
 - gar960 command, 2-1
 - gcdm (decision maker) linker option, 7-37
 - gcov960 invocation, 5-1
 - gdm960

- archive support, 6-11
- displaying archive structure, 6-12
- dumping absolute symbols, 6-5
- dumping archive member contents, 6-14
- invocation, 6-1
- ghist960
 - overview, 14-1
 - invocation, 14-12
- gld960 linker invocation, 7-6
- global uninitialized symbol, IEEE-695, 4-3
- gmung960 command, 9-1
- gnm960 command, 10-1
- grom960, 11-1
 - converting image to hex files, 11-4
 - creating binary images, 11-4
 - invocation, 11-1
 - options summary, 11-2
 - section specification, 11-2
- gsize960 command, 13-1
- gstrip960 command, 15-1

H

- h (Help) archiver option, 2-11
- h (help) linker option, 7-38
- H (sort common symbols) linker option, 7-38
- HP/MRI IEEE 695 object file format, D-1
- hyphen (-) character, use with tools, 1-5

I-J

- i (Insert) archiver option modifier, 2-8
- IEEE-695
 - built-in types, cvt960, 4-5

- converter warning messages, cvt960, 4-8
- object file format, D-1
- ihex rom960 directive, 12-8
- invocation, conventions, 1-4
- J (compress) linker option, 7-39

L

- l (library input) linker option, 7-40
- L (library search path) linker option, 7-39
- l (Local) archiver option modifier, 2-12
- library naming conventions and search order
 - linker, 7-21
- linker, 7-1–7-57
 - B (section start address) option, 7-27
 - binding profile counters to non-standard sections, 7-19
 - c (circular library search) option, 7-29
 - C (startup alternative) option, 7-29
 - calljx, i960® RP processor, 7-19
 - d (define common symbol space) option, 7-31
 - D (inhibit CAVE section compression) option, 7-31
 - default allocation, A-7
 - defsym (define a symbol) option, 7-32
 - directives, A-4
 - e (entry point) option, 7-33
 - ELF/DWARF sections, 7-4
 - environment variables, 7-20
 - f (fill) option, 7-35
 - F (format) option, 7-34
 - G (big-endian target) option, 7-36
- linker (continued)

- gcdm (decision maker) option, 7-37
- h (help) option, 7-38
- H (sort common symbols) option, 7-38
- incremental linking, 7-13
- J (compress) option, 7-39
- l (library input) option, 7-40
- L (library search path) option, 7-39
- library naming conventions and search order, 7-21
- link time optimization, 7-16
- linker directive files, specifying, 7-12
- m (memory map) option, 7-42
- M (multiple definition warning) option, 7-42
- memory block and section allocation, 7-2
- N (name memory map file) option, 7-43
- n (noinhibit output) option, 7-44
- named BSS sections, 7-4
- O (optimization of calls inhibited) option, 7-44
- o (output filename) option, 7-45
- Object Module Format (OMF) compatibilities, 7-14
- options reference, 7-24
- output file, naming, 7-13
- p (position-independence) option, 7-47
- P (profiling) option, 7-46
- R (read symbols only) option, 7-48
- r (relocation) option, 7-49
- S (strip) option, 7-50
- T (section start address) option, 7-27
- t (suppress multiple definition symbol warnings) option, 7-52
- T (target) option, 7-51
- u (unresolved symbol) option, 7-53
- v (verbose) option, 7-54
- V (version) option, 7-54
- v960 (version) option, 7-54
- W (warnings) option, 7-55
- X (compress) option, 7-55
- y (trace symbol) option, 7-56
- Z (program database) option, 7-57
- z (time stamp suppression) option, 7-57
- linker command language, 7-51
 - assignments, A-2
 - expressions and operators, A-2
 - introduction, A-1
- linker directive files, A-35
 - command language, A-1
 - described, 7-4
 - sample, 7-5
 - specifying, 7-12
- linker directives
 - ADDR, A-25
 - ALIGN, A-25
 - CHECKSUM, A-33
 - DEFINED, A-25
 - ENTRY, A-28
 - FLOAT, A-31
 - FORCE_COMMON_ALLOCATION, A-25
 - HLL, A-29
 - INCLUDE, A-32
 - MEMORY, A-5
- linker directives (continued)
 - NEXT, A-25
 - OUTPUT, A-34

- PRE_HLL, A-28
- SEARCH_DIR, A-32
- SECTIONS, A-8
- SIZEOF, A-27
- STARTUP, A-27
- SYSLIB, A-31
- TARGET, A-33
- linker options summary, 7-7
- linking, incremental, 7-13
- lnk960 linker invocation, 7-6

M

- m (memory map) linker option, 7-42
- m (Move) archiver option, 2-13
- M (multiple definition warning) linker option, 7-42
- macro processor, 8-1
 - controlling input, 8-19
 - debugging macros, 8-16
 - diverting output, 8-23
 - including files, 8-22
 - processor options, 8-2
- map rom960 directive, 12-10
- mkfill, 12-12
- mkimage rom960 directive, 12-12
- move rom960 directive, 12-14
- mpp960
 - command, 8-2
 - message prefixes, 8-1

- munger, 9-1
 - options summary, 9-2

N

- N (name memory map file) linker option, 7-43
- n (noinhibit output) linker option, 7-44
- nam960 command, 10-1
- name lister, 10-1
 - options summary, 10-3
- namer tool, 10-1
- names of utilities, 1-3

O

- O (optimization of calls inhibited) linker option, 7-44
- o (Output date) archiver option modifier, 2-14
- o (output filename) linker option, 7-45
- objcopy
 - command, 3-2
 - object converter, 3-1
 - options summary, 3-2
- object file conversion tools, 3-1, 12-1
- Object Module Format (OMF)
 - archives, 2-4
 - compatibilities, 7-14
- optimization, link time, 7-16
- output files, linker
 - naming, 7-13

P

p (position-independence) linker option, 7-47
p (Print) archiver option, 2-14
P (profiling) linker option, 7-46
packhex rom960 directive, 12-15
patch rom960 directive, 12-16
permute rom960 directive, 12-17
profile (p) option, gcov960, 5-3, 5-4

Q-R

R (read symbols only) linker option, 7-48
r (relocation) linker option, 7-49
r (Replace) archiver option, 2-15
rom rom960 directive, 12-18
rom960
 checksum directive, 12-7
 directives summary, 12-4
 ihex directive, 12-8
 invocation, 12-1, 12-3
 map directive, 12-10
 mkfill directive, 12-12
 mkimage directive, 12-12
 move directive, 12-14
 packhex directive, 12-15
 patch directive, 12-16
 permute directive, 12-17
 rom directive, 12-18
 sh directive, 12-21
 split directive, 12-22
rommer, rom960, 12-1

S

S (strip) linker option, 7-50
s (Symbol Table) archiver option modifier, 2-16
section size printer, 13-1
 options summary, 13-2
sh rom960 directive, 12-21
siz960 command, 13-1
slash (/) character, use with tools, 1-5
split rom960 directive, 12-22
statistical profiler, 14-1
 buckets, 14-3
str960 command, 15-1
stripper, 15-1
 options summary, 15-2

T

T (section start address) linker option, 7-27
t (suppress multiple definition symbol warnings)
 linker option, 7-52
t (Table of contents) archiver option, 2-17
T (target) linker option, 7-51
temporary files, archiver, 2-5
test coverage analysis tool, 5-1
 controls summary, 5-2, 5-3
 option summary, 5-3, 5-4
tool names, 1-3
tools, list of, 1-2

U

- u (unresolved symbol) linker option, 7-53
- u (Update) archiver option and modifier, 2-18
- UNIX command line, 1-4
- utilities, list of, 1-2
- utility names, 1-3

V

- v (Verbose) archiver option modifier, 2-19
- v (verbose) linker option, 7-54
- V (Version) archiver option, 2-20
- V (version) linker option, 7-54
- v960 (Version) archiver option, 2-20
- v960 (version) linker option, 7-54

W-X

- W (warnings) linker option, 7-55
- Windows command line, 1-4
- X (compress) linker option, 7-55
- x (Extract) archiver option, 2-21

Y-Z

- y (trace symbol) linker option, 7-56

Z

- Z (program database) linker option, 7-57
- z (Suppress time stamp) archiver option, 2-22
- z (time stamp suppression) linker option, 7-57

