Assembler/Linker/Librarian User's Guide and Reference Manual for the PowerPC Family

Software Version 3.8

Part Number 227463

January 2015

Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: supportnet.mentor.com/

Send Feedback on Documentation: supportnet.mentor.com/doc_feedback_form

Table of Contents

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 7

Chapter 8

List of Examples

List of Figures

List of Tables

Revision History

Revision	Changes	Date
001	Original Issue Manual #101966	1/96
002	Software Update Version 1.3.	10/96
003	Original PC release.	6/97
004	Software Update Version 1.4C. Updated path information. Added 740/750 processor and Win 95.	12/97
005	Product name changes and minor edits.	8/98
006	Software update Version 1.6.	11/98
007	Minor changes.	4/99
008	Updated for software version 1.7.	4/99
009	Updated for software version 1.8.	4/99
010	Updated for software version 1.9.	8/00
011	Updated for software version 2.0. Templates updated for Frame 6.0.	5/03
012	Updated for software version 2.0P	05/04
013	Updated for software version 3.0.	07/04
014	Updated for software version 3.1.	12/04
015	Updated to new template style. Minor edits.	3/05
016	Updated for software version 3.2A.	12/05
017	Updated for software version 3.3	8/07
018	Updated for software version 3.3A	11/07
019	Updated for software version 3.4	3/08
020	Updated for software version 3.5	10/08
021	Updated for software version 3.6	7/09
022	Updated for software version 3.7	8/12
023	Updated for software version 3.7.7	12/12
024	Updated for software version 3.8	1/15

Chapter 1
Introduction

Components of the ASMPPC Package

The Microtec© ASMPPC assembler package consists of an assembler, a linker, and an object module librarian. The assembler package provides an integrated system for developing software applications for the Freescale, AMCC, or IBM PowerPC processor. The following sections describe the components of the Microtec assembler package.

Assembler

The Microtec ASMPPC Assembler converts assembly language programs into relocatable object code. Object modules are suitable for linking with other modules or with libraries of modules.

The ASMPPC Assembler is designed to be similar to the Motorola PowerPC assembler. Many additional directives are provided for versatility. The ASMPPC Assembler processes macros and conditional assembly statements directives. The assembler can generate a cross-reference table as well as a standard symbol table, and the generated code and data can be placed in multiple named sections.

After assembly, the linker links the object modules, which are then suitable for downloading and execution on any supported microprocessor.

Linker

The Microtec LNKPPC Linker combines relocatable object modules into a single absolute object module. Object modules are generated in the System V Release 4 Executable and Linking Format (ELF).

The linker also supports the combining of multiple relocatable object modules into a single relocatable module, which subsequently can be relinked with other modules. This feature is referred to as incremental linking.

If one of the input files is a library of object modules, the linker automatically loads only those modules from the library that are referenced by the other named object modules. The linker produces a link map that shows the final location of all modules and sections and the final absolute values of all symbols. In the XML version of the link map, an interactive call graph can be used to investigate how various global or local symbols are referenced. The text version of the link map contains a cross-reference listing, showing which modules refer to each global symbol.

Execute the linker by specifying which object modules and libraries to use, by specifying a command file on the command line, or both. Any specified object modules and libraries are used in addition to any specified in the command file, or are selected automatically by the linker. The linker reports unresolved external symbols and link errors on the link map or on the terminal.

Object Module Librarian

The Microtec LIBPPC Object Module Librarian lets you maintain a collection of relocatable object modules that reside in one file. Libraries let you automatically load frequently used object modules without concern for the specific names and characteristics of the modules.

Use the librarian to format and organize library files that will subsequently be used by the linker. You can add, delete, replace, and extract modules, as well as obtain a listing of library contents. The Microtec Librarian currently supports the Microtec IEEE-695 library format and the SYS V PORTAR library format. The default is to produce IEEE-695 libraries.

ASMPPC Features

Significant features of ASMPPC include:

- Support for PowerPC 4xx, 5xx, 6xx, 7xx, 8xx, 5xxx, 74xx, 82xx, 83xx, 85xx, and 86xx series of microprocessors.
- Support for PowerPC Floating-Point Unit (FPU), Altivec, and Signal Processing Engine APU (SPE APU) instructions
- Support for nonextended (static) PPC EABI with DWARF 2.0 debug information
- Support for symbolic constants
- Assembly-time relative addressing
- Data creation statements
- Storage reservation statements
- Specified character strings in ASCII
- Flexible assembly listing control statements
- Symbolic and relative address assignments and references
- Symbol and cross-reference table listings
- Object modules in extended AT&T System V Executable and Linking Format (ELF)
- Relocatable (incremental linking) object modules
- Support for forward references

- Conditional assembly facility
- User-defined macro facility
- EDGE™ Debugger information
- Support of case-sensitive symbols
- Support for Microtec IEEE-695 and ELF PORTAR library archives

These features will help you produce well documented, modular, working programs in a minimum of time.

Microtec Toolkit

The Microtec ASMPPC assembler package is part of a larger system of embedded software development tools called the Microtec Toolkit. The following are other components of the Microtec Toolkit for the PowerPC family.

C Compiler

The Microtec MCCPPC C Compiler converts both ANSI C and traditional C source programs into tight, efficient assembly language code for the ASMPPC Assembler. You can use the C compiler to create ROMable programs and produce debugging information for your debugger.

C++ Compiler

The Microtec CCCPPC C++ optimizing cross compiler converts C++ source programs into tight, efficient assembly language code for the ASMPPC Assembler. The C++ compiler can also produce information necessary for debugging with your debugger.

The CCCPPC compiler supports all the features of the MCCPPC C compiler, as well as C++ language features described in *The Annotated C++ Reference Manual*, such as templates, multiple inheritance, overloading resolution, memberwise assignment and initialization, static member functions, abstract classes with pure virtual functions, pointers to members, and type-safe linkage. By default, an executable built with CCCPPC uses the C++ libraries, even if the source code is composed entirely of C files.

EDGE Debugger

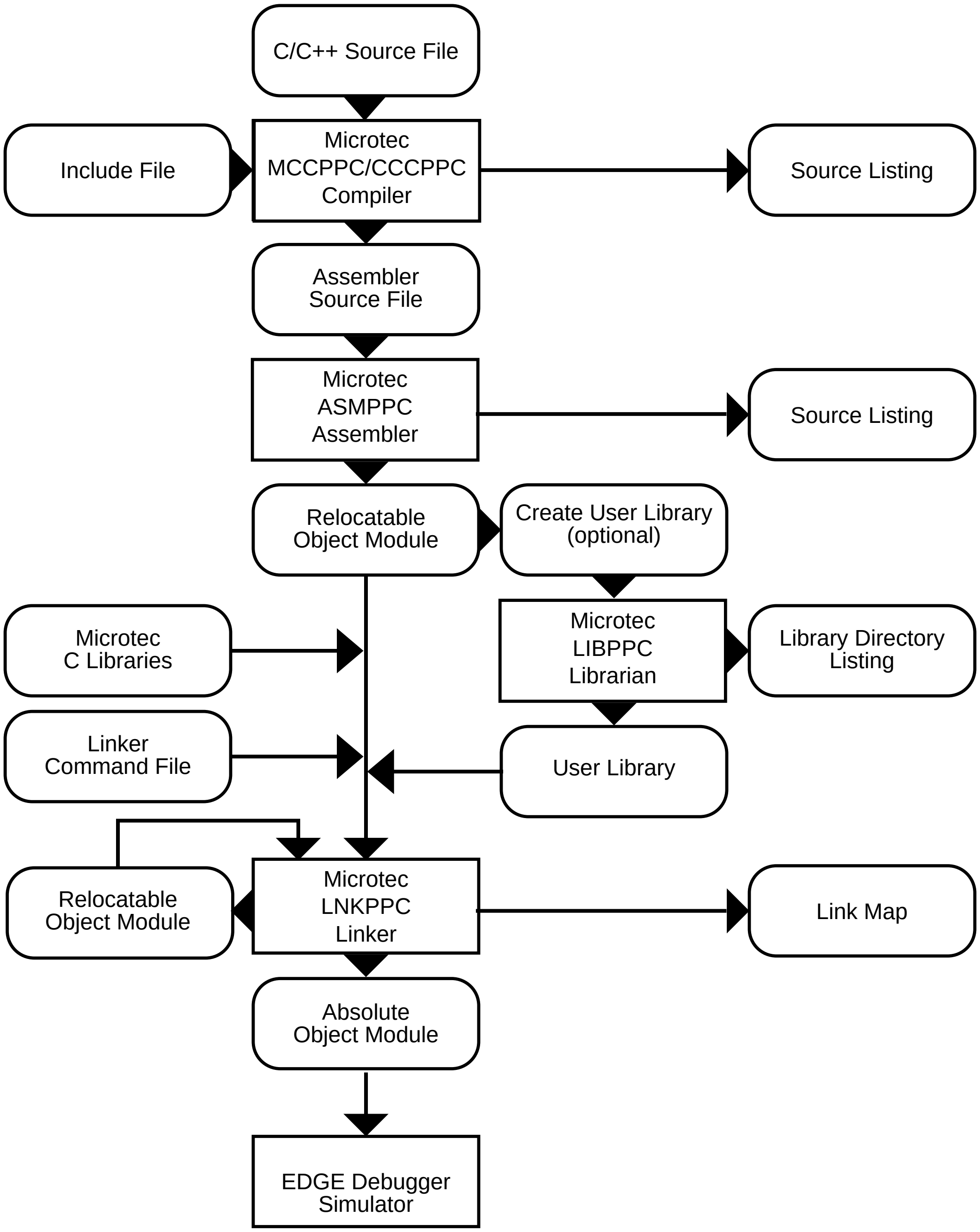
The EDGE Debugger lets you monitor and control the execution of programs at the source level using a window-oriented user interface. You can examine or modify the value of program variables, procedures, and addresses using the same source-level terms, definitions, and structures defined in the original source code. The interactive debugger gives you complete control of the program through an execution environment, such as a simulator.

The EDGE Debugger uses a powerful command language that allows simple and complex breakpoint setting, single-stepping, and continuous variable monitoring. Input and output can be directed to or from files, buffers, or windows. In addition, a sophisticated macro facility allows complex command sequences to be associated with events, such as the execution of a specific statement or the accessing of a specific data location. These features let you isolate errors and patch your source code. For more details, consult the EDGE documentation on codelets.

Data Flow

Figure 1-1 shows the components of the Microtec ASMPPC assembler package and the ways in which they communicate with each other and with the rest of the Microtec Toolkit.

Figure 1-1. Components of the Microtec Toolkit for the PowerPC Family



Chapter 2
Command Usage

This chapter describes the basic use of the assembler, linker, and object module librarian for use on all hosts running the UNIX or Windows operating systems.

The chapter is divided into sections, one for each tool; each section contains descriptions of the following information:

- Invocation syntax
- Environment variables, if any
- Input and output file name defaults
- Command line flag descriptions, if any
- Invocation examples

The last section of the chapter describes the return codes generated by the tools.

Examples in this chapter are identified by operating system name. The name UNIX refers to any UNIX or UNIX-like operating system, such as Solaris. The name Windows refers to an MS-DOS Command Prompt window running under Windows XP, Windows Vista, or Windows 7. The tools do not run under native MS-DOS.

ASMPPC Assembler

The Microtec ASMPPC Assembler produces object code, typically for later use by the LNKPPC Linker and LIBPPC Librarian.

Invocation Syntax

The following command line syntax shows how to invoke the ASMPPC Assembler. An invalid command line entry generates an error message, which is written to the standard error output device. On UNIX and Windows systems, the standard error output device is **stderr**. You may use the backslash character (\) to continue a command line entry on the next line.

```
asmppc [-d options_file] [-D sym[=value]] [-E] [-f flag_list] [-g] [-h]
[-I include_dir] ... [-k] [-l[list_file]] [-o object_file] [-p proc] [-
plist] [-Q opt] [-s] [-V] [-w] [source_file]
```

where:

asmppc

Invokes the assembler.

-d*option_file*

The **-d** option directs the assembler to read command line options from the specified file.

-D*sym*[=*value*]

Defines the symbol *sym* at assembly time. The parameter *value* must be a constant and an integer. If *value* is not specified, *sym* is set to 1. This option has the same effect as the **.set** directive.

-E

Causes the object file to be generated in little-endian mode. The default is to generate a big-endian object file.

-f *flag_list*

Enables or disables the internal assembler listing control flags (see Table 2-1). The **-l** option must also be specified in order for many of these flags to be enabled.

-g

Causes the assembler to generate high-level debug information for assembly source files. Line information will be emitted for each PowerPC instruction used outside of a macro or repeat block. The assembler will also emit symbolic information for local and global labels and variables. This option allows viewing an assembly source file in a high-level source window in debuggers. It should only be used on assembly source files having a single section containing executable code; otherwise, no mechanism exists for representing line information for instructions outside of that section. This option also should not be used on a file that contains high-level debug information generated by Microtec C or C++ compilers.

-h

Displays usage information.

-I *include_dir*

Adds the specified directory to the list of directories to search for **.include** files with relative path names. The assembler first searches for **.include** files in the present working directory and then in the directories specified with the **-I** option, if any. The **-I** option may be specified any number of times.

-k

Causes the object file to be retained even if an error has occurred during the processing of the source file. Normally, this file is removed if an error is detected. Using this option may result in an invalid object file, so care should be taken.

-l[list_file]

Indicates that a listing file will be written to *list_file*, if present. If *list_file* is not specified, a listing file will be written to the standard output device. The standard output can be redirected to a file. No spacing is permitted between -l and *list_file*.

-o *object_file*

Overrides the default object module filename with the specified filename. The default object filename is the *source_file*; the extension listed in Table 2-2 replaces the extension of *source_file* and the output file is placed in the current directory.

-plist

Generates a list of PowerPC variants supported by the Microtec toolkit. This list is sent to the standard output. Do not use other options or arguments with the **-plist** option. **-P** has been deprecated.

-p *proc*[/*modifier*]

Specifies the target processor upon which the source executes. Allowable processor types are: **401gf, 403ga, 403gb, 403gc, 403gcx, 405cr, 405ep, 405gp, 405gpr, 440ep, 440gp, 440gx, 505, 509, 5121e, 5123, 5200, 5200b, 533, 534, 535, 536, 555, 5553, 5554, 556, 560, 561, 562, 563, 564, 565, 566, 5xx, 603, 603e, 603e6, 603e7v, 603r, 604, 604e, 740, 7400, 7410, 7410t, 7441, 7445, 7445a, 7447, 7447a, 7448, 745, 7450, 7451, 7455, 7455a, 7457, 745b, 750, 750cx, 750cxe, 750cxr, 750fl, 750fx, 750gl, 750gx, 750gx_dd11_8, 750l, 755, 755b, 755c, 801, 821, 823, 823e, 8240, 8241, 8245, 8245a, 8247, 8248, 8250, 8250a, 8255, 8255a, 8260, 8260a, 8264, 8264a, 8265, 8265a, 8266, 8266a, 8270, 8270vr, 8271, 8272, 8275, 8275vr, 8280, 8313, 8313e, 8314, 8314e, 8315, 8315e, 8321, 8321e, 8323, 8323e, 8343, 8343e, 8347, 8347e, 8349, 8349e, 8358e, 8360e, 8377e, 8378e, 8379e, 850, 850de, 850dsl, 850sr, 852t, 853t, 8533, 8533e, 8540, 8541e, 8543, 8543e, 8544, 8544e, 8545, 8545e, 8547e, 8548, 8548e, 8555e, 855t, 8560, 8567, 8567e, 8568, 8568e, 8572, 8572e, 857dsl, 857t, 859dsl, 859p, 859t, 860, 860de, 860dp, 860dt, 860en, 860p, 860sr, 860t, 8610, 862, 862p, 862t, 8640, 8640d, 8641, 8641d, 866, 866p, 866t, 870, 875, 880, 885, 8xx, com, e300, e300c2, e300c3, e500v1, e500v2, ec603e, ec603e6, ec603e7v, em603e, g2, g2_le, npe405h, npe405l,** and **ppc**. The /F modifier can be used to indicate that all floating-point instructions are executable on a processor, possibly through software simulation. The /NF modifier indicates that no floating-point instructions are available. If neither modifier is used, the assembler accepts any instructions that are executable in hardware for the indicated processor. The default processor type is 603.

-Q *opt*

Affects how error and warning messages are emitted by the assembler. The allowable options are:

- **ms A<number>[[,] A<number>]**
  When this option is used, listed messages are ignored by the assembler and are not counted in the error/warning summary. Non-fatal messages have a **-D** following the error number in the message. All other errors are considered fatal and cannot be ignored.
- **s**
  Suppresses the error/warning summary at the end of the listing.
- **e**
  Suppresses all discretionary error and warning messages. Any errors that do not have a **-D** following the error number will continue to display and be counted.
- **w**
  Suppresses all warning messages. This is equivalent to the **-w** option.

-s

Strips all DWARF 2 debug information from the object file. This option may be useful if an assembly source file contains DWARF 2 information but there is no need to debug the resulting object file. This option does not affect non-debug sections or contents.

-V

Lists the current version number to the standard output device and exits the assembler. All other command line entries are ignored. The version number is always displayed in the output listing. (default: no version number displayed on standard output)

-w

Suppresses warning messages in the assembler output listing and the standard output.

source_file

Specifies the assembler source file. The default filename extension is **.s** (UNIX) and **.src** (Windows). The source file must be the last entry on the command line, excluding redirected standard output listings. Only one *source_file* is permitted on a command line. A fatal error occurs if a file name is not supplied for *source_file* and no source file is redirected as input to the assembler.

If you do not specify any options, the following defaults will be used:

- No listing file
- *source_file***.o** (UNIX) or *source_file***.obj** (Windows) in Executable and Linking Format (ELF) format in the current directory

Refer to Table 2-1 for the default flags that will be used.

Assembler Command Flags

Table 2-1 lists the assembler command flags for *flag_list* in the **-f** *flag_list* option. With the **-f** *flag_list* option, internal assembly controls are enabled and disabled. Command line flags that affect the output listing have no effect unless the **-l** option has been specified.

Some flags may be negated by using the prefix **n** (for example, **nc** negates the **c** flag). This negation only applies to the next flag; multiple **n** prefixes may be required. If multiple items are indicated, double quotes (**" "**) or single quotes (**' '**) must be used to enclose the items. If an unrecognized flag is used, if a value is missing, or if a flag is negated that takes a value, the assembler issues an error message and aborts.

Flags specified on the command line can override other directives in the source file. More information can be obtained about the flags under the **.lflags** directive in Chapter 6, "Assembler Directives". The **-f** *flag_list* option has the same effect as the **.lflags** directive in the beginning of a source file.

Note that if the same flag is indicated multiple times on the command line, the value or state of the last flag is used.

Table 2-1. Assembler -f Option Command Flags

Flags	**Meaning (positive form)**
a	Causes automatic program counter alignment for data directives. (default: **a**)
b*value*	Sets the default alignment for common variables. (default: **b**4 or **b**8, depending on the selected PowerPC variant; use the **v** flag to determine the default size)
c	Lists instructions not assembled due to conditional assembly statements. (default: **nc**)
d*value*	Sets the maximum macro recursion depth. (default: **d**100 for UNIX, **d**25 for Windows)
f	Causes additional information to be emitted when used with the **v** flag. (default: **nf**)
i	Lists **.include** files on the program listing. (default: **ni**)

Table 2-1. Assembler -f Option Command Flags (cont.)

Flags	Meaning (positive form)
k	Causes all local Microtec C/C++-style compiler-generated labels to be retained. (default: **nk**)
l*value*	Sets the line length of the output listing to the value specified. The line length must be between 40 and 4096. (default: **l**132)
m	Lists macro and repeat expansions on the program listing. (default: **nm**)
o	Lists data storage overflow. Normally, only the first 4 bytes are displayed in the output listing. (default: **no**)
p*value*	Sets the page length of the output listing to the value specified. The minimum page length is 20; 0 will turn off pagination. (default: **p**55)
r	Allows the matching of predefined register symbols in expressions. (default: **r**)
s	Lists the symbol table on the output listing. This flag is overridden by the **x** flag. (default: **ns**)
v	Emits a list of instructions and registers that are recognized and accepted by the selected PowerPC variant. This list is emitted to standard output after any other assembler output is produced. Use the **f** flag to generate additional content. (default: **nv**)
w	Wraps listing lines that exceed the maximum line width. (default: **w**)
x	Lists the cross-reference table on the output listing. This flag overrides the **s** flag. (default: **nx**)

Filename Defaults

If *source_file* does not contain an extension, the assembler assumes the extensions listed in Table 2-2. An output object module is produced by default, but if no output filename is specified on the command line, the assembler uses *source_file* as the root filename and appends the default filename extensions listed in Table 2-2.

Table 2-2. Assembler Default Filename Extensions

File	Default UNIX Extension	Default Windows Extension
Assembler source file	*.s*	*.src*
Relocatable object file	*.o*	*.obj*

Filenames can include full file specification information including:

- Pathname
- Filename
- Extension

If you do not supply a pathname, filename, or extension, the current defaults are used.

Invocation Examples

The examples that follow show how to enter various options on the command line.

Example 2-1. Assembler Invocation

```
asmppc -l -f x -o multd.o multd.s > multd.l
```

This command reads the input source from the file **multd.s**. The output listing is written to **multd.l**. The output object module is written to **multd.o**. The **-f** option **x** flag generates the cross-reference table in the listing. If **-l** had not been specified, no listing would have been produced.

```
asmppc -o temp.o divd.s
```

The input source **divd.s** is assembled. The output listing is not produced because the **-l** option is not used on the command line. The output object module is named **temp.o**.

LNKPPC Linker

The Microtec LNKPPC Linker combines relocatable object modules and libraries into a single absolute or relocatable module. In the process, it resolves memory and external references.

Invocation Syntax

The following command line syntax shows how to invoke the LNKPPC Linker. Note that the linker must be invoked with a command file or an input object file. Use the line continuation character (\) to continue the command line entry on the next line.

```
lnkppc [-c command_file] [-d option_file] [-e entry_name] [-f flag_list]
         [-g] [-h] [-i] [-K{0|e|f}] [-m[mapfile]] [-o output_objectfile]
         [-p variant] [-[n]q] [-Q opt] [-r] [-s]
         [-u external_name] [-v] [-V] [-Vi] [-x] [-Zb[012]] [-Z[n]s]
         [-Z[n]x[number]] [input_objectfile [input_objectfile] ...]
```

where:

lnkppc

Invokes the linker.

-c *command_file*

Identifies the named file as a command file, which contains one or more linker commands. If the name is not an absolute path, the command file is searched for relative to the current directory, or, if not found there, then relative to the **../lib** directory from where the linker executable is located.

-d*option_file*

The **-d** option directs the linker to read command line options from the specified file.

-e *entry_name*

Specifies the name of the entry point and overrides the **__mri_start** entry point name existing in the object modules.

-f *flag_list*

Specifies that subsequent entries are flags that are used to enable or disable various internal options (see Table 2-4).

-g

Creates a command file that contains linker public commands for all global symbols and their values. This command file will have the same name as the output file, but with a **.glb** extension. This option can only be used with the creation of an absolute file.

-h

Displays linker usage information and exits.

-i

Specifies that the output object module produced is relocatable instead of absolute (absolute is the default). Normally, this output module is one step in an incremental link. The relocatable output file will be in ELF format.

-K{0|e|f}

Sets the minimum error severity level required to keep the output file.

-K0 (default) Keeps output file only if no errors occurred

-Ke When possible, keeps output file even if non-fatal errors occurred, this is equivalent to the -fnr option

-Kf When possible, keeps output file even if fatal errors occurred

-m[*map_file*]

Indicates that a map file will be written to *map_file*, if present. If *map_file* is not specified, a map file is created with the same name as the output file, but with an *.xml* extension. The default format for the map file is an interactive XML file, that can be viewed from compatible browsers, such as Internet Explorer. A textual version of the map file can be created by specifying a map file name with a *.map* extension. No spacing is permitted between **-m** and *map_file*.

-[n]q

The compiler normally generates a **.libinfo** section in all object files. With **-q** (default is on), the linker uses this information to select the default run-time libraries. The **-nq** option can be used to cause the linker to ignore the **.libinfo** sections. The **-nq** option also prevents a default command file from being used, if a command file is not indicated with the **-c** option.

-o *output_object_file*

Identifies the named file (*output_object_file*) as the output object module. If you do not enter an output object file name, the output object file will have the root name of *command_file* or *input_object_file* if there is no command file (*input_object_file* is the name of the first input object file specified on the command line). The default location for the output file will be the current directory.

By default, the output object module is absolute in the ELF format with the extension **.x** (UNIX) or **.abs** (Windows). With the **-i** option, the output object module is relocatable in the ELF format with the extension **.o** (UNIX) or **.obj** (Windows).

-p *proc*

Specifies the target processor upon which the linker executes. This is only used to select a default command file (see **-q**). Allowable processor types are: **401gf, 403ga, 403gb, 403gc, 403gcx, 405cr, 405ep, 405gp, 405gpr, 440ep, 440gp, 440gx, 505, 509, 5121e, 5123, 5200, 5200b, 533, 534, 535, 536, 555, 5553, 5554, 556, 560, 561, 562, 563, 564, 565, 566, 5xx, 603, 603e, 603e6, 603e7v, 603r, 604, 604e, 740, 7400, 7410, 7410t, 7441, 7445, 7445a, 7447, 7447a, 7448, 745, 7450, 7451, 7455, 7455a, 7457, 745b, 750, 750cx, 750cxe, 750cxr, 750fl, 750fx, 750gl, 750gx, 750gx_dd11_8, 750l, 755, 755b, 755c, 801, 821, 823, 823e, 8240, 8241, 8245, 8245a, 8247, 8248, 8250, 8250a, 8255, 8255a, 8260, 8260a, 8264, 8264a, 8265, 8265a, 8266, 8266a, 8270, 8270vr, 8271, 8272, 8275, 8275vr, 8280, 8313, 8313e, 8314, 8314e, 8315, 8315e, 8321, 8321e, 8323, 8323e, 8343, 8343e, 8347, 8347e, 8349, 8349e, 8358e, 8360e, 8377e, 8378e, 8379e, 850, 850de, 850dsl, 850sr, 852t, 853t, 8533, 8533e, 8540, 8541e, 8543, 8543e, 8544, 8544e, 8545, 8545e, 8547e, 8548, 8548e, 8555e, 855t, 8560, 8567, 8567e, 8568, 8568e, 8572, 8572e, 857dsl, 857t, 859dsl, 859p, 859t, 860, 860de, 860dp, 860dt, 860en, 860p, 860sr, 860t, 8610, 862, 862p, 862t, 8640, 8640d, 8641, 8641d, 866, 866p, 866t, 870, 875, 880, 885, 8xx, com, e300, e300c2, e300c3, e500v1, e500v2, ec603e, ec603e6, ec603e7v, em603e, g2, g2_le, npe405h, npe405l,** and **ppc**.

-Q *opt*

Affects how error and warning messages are emitted by the linker. The allowable options are:

- **me A<number>[[,] A<number>]**
  When this option is used, warning messages are upgraded to errors.
- **ms A<number>[[,] A<number>]**

  When this option is used, listed messages are ignored by the linker and are not counted in the error/warning summary. Non-fatal messages have a **-D** following the error number in the message. All other errors are considered fatal and cannot be ignored.
- **s**
  Suppresses the error/warning summary at the end of the listing.
- **e**
  Suppresses all discretionary error and warning messages. Any errors that do not have a **-D** following the error number will continue to display and be counted.
- **i**
  Suppresses all informational messages.
- **w**
  Suppresses all warning and informational messages.

-r

Causes run-time relocation information to be generated in the **.prelude** section. If the program is run at a location different from that specified during the link process, then at startup, the program uses the **.prelude** data to locate all address references within the program and correct them. The function that updates these address references is named __StaticPrelude__ and is in the runtime library source file *prelude.c*. This option does not generate relocation information for the initdat section (created when using the INITDATA linker command) so is not recommended for use with INITDATA.

-s

Causes the removal of all debug and symbol information from the object file created by the linker. This option can be used to prevent the creation of symbol tables, debug sections, and string information that would normally be emitted into the output file. Any object file created through the use of this object cannot be debugged at the language level.

-u *external_name*

Forces the loading of the module that defines the *external_name* in a library. A message will be generated if the symbol cannot be resolved. This option cannot be used to resolve external variables.

-v

Generates a list of supported PowerPC variants. This list indicates which PowerPC variants are considered the same and which variants may be linked together.

-V

Displays the version number of LNKPPC and then exits.

-Vi

Displays the command file and libraries selected by the linker (see **-q**).

-x

Merges **.sdata2** sections of type “read-only” and “read-write” into one “read-write” section. The default behavior is not to combine these sections.

-Zb{0|1|2}

Causes the creation of a section, called **.PPC.EMB.lilypond**, that contains code to branch to a fixed location anywhere in the 32-bit address space. Normally, code may only branch to another location that is within 32 megabytes, plus or minus, of the current location. The **-Zb1** and **-Zb2** options cause a branch to the destination to be created automatically, which is placed in the **.PPC.EMB.lilypond** section. Any branch to a given destination that results in a relative branch that exceeds 32 megabytes is modified by the linker to refer to the linker-created code instead. This results in an additional one or two branches, but the destination is reachable without rewriting or recompiling code. The **.PPC.EMB.lilypond** section should be placed in either the low or high 32 megabytes of memory, so locations within that section can be accessed through an absolute branch instruction.

The **-Zb0** option, which is the default, suppresses the creation of the **.PPC.EMB.lilypond** section. When this option is in force, any branch that results in a relative offset greater than 32 Megabytes will result in a link time error. This code must be modified before it will execute correctly.

The **-Zb1** option causes the creation of a four-instruction block of code that may be used to reach any address within the PowerPC 32-bit address space. The **CTR** Special Purpose Register is modified to hold the destination address. One four-word block (or **lilypad**) is created for each location that is branched to. Multiple references to the same destination share the same four-word block. If a branch results in a relative offset that is greater than 32 megabytes, that instruction is modified into an absolute branch (**BA**) instruction that goes to the appropriate four-word block. That lilypad then modifies the **CTR** SPR and branches to the final destination. The result of using such a lilypad is the execution of an additional 4 instructions, including an additional branch. If a branch does not require a lilypad, then one is not used. The only performance penalty is that a created lilypad may go unused.

The **-Zb2** option behaves similarly to the **-Zb1** option, except each created lilypad is instead three instructions in size. A two-instruction block of code, at the end of the **.PPC.EMB.lilypond** section, is then shared by all of the lilypads. This option may save a little space, but it adds the overhead of an additional instruction and branch.

-Z[n]s

Causes the output object file's string table to be optimized. All symbols from input files are placed in the string table. If the **-Zs** option is used, symbols will only be added to the string table if the symbol cannot already be found in the string table. This can occur if the symbol is either a local symbol and therefore exists in more than one input file, or if the symbol matches the end of another symbol previously placed in the string table (symbols in the string table are terminated with a null character, or **\0**). If the **-Zns** option is used, it will cause symbols to always be added to the string table, without any regard for whether the symbol already exists there, either as a complete symbol or part of a different symbol. This option only affects the size of the string table and the resulting executable size. There is no effect on execution or the ability to debug the executable. Note that section names will only appear once in the string table, regardless of the setting of this option. Turning on string table compression will result in longer link times. The default operation of the linker is to always add symbols to the string table (**-Zns**).

-Z[n]x[*number***]**

Indicates the amount of iteration that should occur for the removal of duplicate non-inlined copies of inlined functions. If no value is indicated, the linker will iterate until no more duplicates are detected. If a number is indicated, then that will be the maximum number of iterations that the linker will perform while checking for duplicates. It is possible that the linker may stop iterating before that number of iterations is reached. If the 'n' modifier is used, no iterations will occur.

input_object_file

Specifies an input object module or library. The default file name extension is **.o** (UNIX) or **.obj** (Windows). Both ELF PORTAR and Microtec IEEE-695 libraries are supported as input objects. The linker can also accept input files that have been compressed with the **gzip** utility. Any object or library file with a **.gz** extension is considered to have been compressed. The linker will automatically decompress such a file and use it as if it were stored normally. If the name of the input file is not an absolute path, the file is looked for relative to the current directory or, if not found there, relative to the **../lib** directory from where the linker executable is located.

Environment Variables

The LNKPPC linker uses the **TMPDIR** (UNIX) or **TMP** (Windows) environment variable for determining where to create temporary files. If the appropriate environment variable is not present or if the indicated directory is not usable, the linker will give an error message. If you use these variables, they must be set before invoking the linker.

Filename Defaults

If file name extensions are not specified, the linker assumes the file name extensions listed in Table 2-3. If the output object file is not specified on the command line with the **-o** option, the linker uses the command file name or the first input object file name as the root file name and appends the default file name extensions listed in Table 2-3.

Table 2-3. Linker Filename Extensions

File	Default UNIX Extension	Default Windows Extension
Input command file	.cmd	.cmd
Input ELF relocatable object file	.o	.obj
Output ELF relocatable object file	.o	.obj
ELF absolute object file	.x	.abs

The linker uses the current directory if you do not specify a directory.

Linker Command Flags

Command line flags are used to control certain internal linker options. Table 2-4 lists the legal flags. Precede the flag with the letter **n** to disable it. Flag entries can be separated by commas (**,**). If commas are used to separate entries, all of the entries must be enclosed by double quotes (**" "**) or single quotes (**' '**) on the command line.

Table 2-4. Linker Command Flags -f Option

Flag	Meaning (Positive Form)
c	Includes the external symbol cross-reference table in the text map file listing. This has no effect on the XML map file listing. (default: **c**)
e	Turns on alignment checking for PowerPC relocation types. (default: **ne**)
f	Causes the contents of all sections to be held in memory. If this option is not used, sections will be read from input files as needed. This option should only be used if enough virtual memory exists to hold the entire resulting executable. (default: **nf**)

Table 2-4. Linker Command Flags -f Option (cont.)

Flag	Meaning (Positive Form)
i	Causes common symbols to be created in the same order in which they are read from input files. (default: **ni**)
o	Produces an output object module. (default: **o**)
p	When used with the **-s** option, causes the symbol table to be emitted into the output file while allowing the debug sections to be removed. (default: **np**)
r	Causes the output object file and the global symbol output object file (if **-g** was specified) to be removed in the event of a non-fatal error condition. Normally, these files are retained unless a fatal error occurs. (default: **r**)
s	Causes common symbols to be placed in the **.sbss** section if the common symbol is accessed using a Small Data Area relocation form. (default: **ns**)
t	Includes the local symbol table in the text map file listing. This has no effect on the XML map file listing. (default: **t**)
u	Suppresses any warnings or informational messages from the linker or other consuming tools that would result from linking third-party object files with object files generated by Mentor Graphics tools. (default: **nu**)
x	Includes the external definition symbol table in the text map file listing. This has no effect on the XML map file listing. (default: **x**)
z	Causes compressed input object files and libraries to be stored temporarily in an uncompressed form during linking. If this option is not used, input files may be decompressed several times. This option should only be used if there is enough temporary space to hold all input files in their decompressed states. This option can be useful in reducing network traffic. (default: **nz**)

Invocation Examples

The following examples show various methods of invoking and using the linker.

Invoking the Linker With a Command File

From the command line, the linker lets you specify a command file and optionally, additional object modules. Any object module files specified on the command line will be named in a

LOAD command. This **LOAD** command will be inserted before the first **LOAD** command in the outermost command file (**LOAD** commands in included command files are not checked).

Command files let you set section addresses, define section loading order, and load a default set of object modules and libraries. Command files enter linker commands in batch mode.

Example 2-2. Using a Command File With the Linker

```
lnkppc -c sample.cmd -o sample.x -f nc -msample.map
```

The linker reads the command file **sample.cmd** and produces a link map, which is redirected (written) to the file **sample.map**. The output object module produced is **sample.x**. The **nc** flag suppresses the printing of the external symbol cross-reference table in the text map file listing.

```
lnkppc -c sample.cmd -msample.map mod1.o mod2.o mod3.o
```

The linker reads the command file **sample.cmd**, inserts three **LOAD** commands that load the three object modules **mod1.o**, **mod2.o**, and **mod3.o**, and executes the commands in the file. The output object module is named **sample.x**, and the output map file is **sample.map**.

If the file **sample.cmd** contains the following code:

```
; Load modules test1.o, test2.o and any modules from
; lib1.lib that resolve any remaining external
; references.
LOAD test1.o
LOAD test2.o
LOAD lib1.lib
```

the command file processed by the linker will be:

```
LOAD mod1.o
LOAD mod2.o
LOAD mod3.o
LOAD test1.o
LOAD test2.o
LOAD lib1.lib
```

Invoking the Linker Without a Command File

If no command file is specified, the linker selects one, based on either the processor selected with the **-p** option or based on information in the input object files. If no command file is desired, the **-nq** option prevents one from being selected, as well as prevents any default libraries from being used.

If no command file is indicated or selected, the linker constructs a command file consisting of LOAD commands for the object module files and libraries listed on the command line:

```
lnkppc -nq -o sample -i -msample.map mod1.o mod2.o mod3.o
```

The linker loads the three object modules **mod1.o**, **mod2.o**, and **mod3.o**. The **-i** option generates **sample.o**, a relocatable object module for incremental linking. The link map is redirected (written) to the file **sample.map**.

The command file processed by the linker will be:

```
LOAD mod1.o
LOAD mod2.o
LOAD mod3.o
```

LIBPPC Librarian

The Microtec LIBPPC Object Module Librarian builds and maintains program libraries. Libraries are collections of relocatable object modules residing in a single file. The LIBPPC Librarian reads and stores objects in either the Microtec IEEE-695 library format or the SYS V PORTAR library format. Newly-created libraries are saved in the Microtec IEEE-695 library format unless commands are used to specify the SYS V PORTAR library format.

Invocation Syntax

The command line can be continued on the next line if a backslash (\) is the last character on the line to be continued (on UNIX systems only or as allowed by the command interpreter).

Only certain library functions can be specified on the command line: **ADDMOD**, **DELETE**, **REPLACE**, **EXTRACT**, **DIRECTORY**, **FULLDIR**, and **FORMAT**. The **ADDMOD**, **DELETE**, **REPLACE**, and **EXTRACT** commands are processed in the same order that they appear on the command line. The **DIRECTORY** and **FULLDIR,** and **FORMAT** commands are applied after the other commands are executed. If both a **DIRECTORY** and a **FULLDIR** option are specified, only the **FULLDIR** command is invoked. Only certain multiple object modules and files used as arguments to those commands are separated by commas (**,**). If commas are used as separators, or if file names contain spaces, double quotes (“ ”) or single quotes (‘ ’) must be used to enclose each list.

The syntax is as follows:

```
libppc [-f l | -f s] [-Q<opt>] [-a a_lst] [-d d_lst]
     [-r r_lst] [-e e_lst] [-E] [-c] [-p] [-t] lib_file
libppc [-Q<opt>] < cmd_file
libppc [-Q<opt>]
libppc [-h]
libppc [-V]
```

where:

libppc

Invokes the librarian.

-a *filename*[,*filename*]...

Adds the named object files to the library. This option is equivalent to the **ADDMOD** librarian command. The standard syntax is shown with commas between file names. Wildcard characters can also be used.

-c

Causes any written library to be written in the Microtec IEEE-695 library format. This is the default behavior for created libraries. For existing libraries, the librarian will preserve the original library format unless specified to change it. This option may be used to convert SYS V PORTAR format libraries to the Microtec IEEE-695 format. This option is equivalent to the FORMAT IEEE librarian command.

-d *module_name*[,*module_name*]...

Deletes the named module(s) from the library. This option is equivalent to the **DELETE** librarian command.

-e *module_name*[,*module_name*]...

Extracts the named module(s) from the library into the current directory. Specifically, the module(s) are copied to a file having the same name as *module_name*. The modules are not deleted from the library. This option is equivalent to the **EXTRACT** librarian command.

-E

Causes all of the modules in the indicated library to be extracted to the current directory. Each module will be stored in its own object file. A warning will be displayed if the library contains no modules. This option is equivalent to the **EXTRACTALL** librarian command.

-f l

Lists the entire symbol table and module names on the standard output device. The standard output can be redirected to a file. This option implies the **-f s** option since it lists module names in addition to listing the entire symbol table. This option is equivalent to the **FULLDIR** librarian command.

-f s

Lists only the module names in the library on the standard output device. This option is equivalent to the **DIRECTORY** librarian command.

-h

Displays usage information.

-p

Causes any written library to be written in the SYS V PORTAR library format. For existing libraries, the librarian will preserve the original library format unless specified to change it. This option may be used to convert Microtec IEEE-695 format libraries to the SYS V PORTAR format. This option is equivalent to the FORMAT PORTAR librarian command.

-Q *opt*

Affects how error, warning, and informational messages are emitted by the librarian. The allowable options are:

- **ms A<number>[[,] A<number>]**
  When this option is used, listed messages are ignored by the librarian and are not counted in the error/warning summary. Non-fatal messages have a **-D** following the error number in the message. All other errors are considered fatal and cannot be ignored.
- **s**
  Suppresses the error/warning summary at the end of the listing.
- **e**
  Suppresses all discretionary error, warning, and informational messages. Any errors that do not have a **-D** following the error number will continue to display and be counted.
- **i**
  Suppresses all informational messages.
- **w**
  Suppresses all warning and informational messages.

-r *filename*[,*filename*]...

Replaces the named module(s) in the library with object files having the same name. Wildcard characters can also be used. This option is equivalent to the **REPLACE** librarian command.

-t

Preserves the original dates when extracting object files. This option is only useful for SYS V PORTAR libraries.

-V

Displays the librarian version number and a copyright notice. The librarian then exits.

library_filename

Specifies the library to be read or written.

Output from the librarian will be displayed on your terminal unless you redirect your output to a file.

Environment Variables

The LIBPPC Librarian uses the **TMPDIR** (UNIX) or **TMP** (Windows) environment variable for determining where to create temporary files. If the appropriate environment variable is not present or if the indicated directory is not usable, the librarian will give an error message. If you use these variables, they must be set before invoking the librarian.

Filename Defaults

A summary of librarian filename default extensions is shown in Table 2-5.

Table 2-5. Librarian Filename Extensions

File	Default UNIX Extension	Default Windows Extension
Library file	.lib	.lib
Backup Library File	none	.bak
Listing file (as output from the internal librarian **FULLDIR** command)	.lst	.lst
Object File	.o	.obj

The listing file contains output from the internal librarian **FULLDIR** command.

Default file extensions are assumed when you do not append an extension. The librarian looks for a period (**.**) in the filename, scanning from right to left, and then compares the extension found to the default (**lib**). If they are not the same, the librarian reports an error.

Example 2-3. Library Filenames

If the filename is **lib.lib**:

open lib.	(This command fails.)
open lib.l	(This command fails.)
open lib.lib	(This command succeeds.)
open lib	(This command succeeds.)

Invocation Examples

The LIBPPC Librarian can be invoked in interactive mode, command line mode, and command file mode. Examples of these methods are described in the following sections.

Interactive Mode

You can enter librarian commands interactively from the terminal. A help facility is available by typing **h** or **help**.

Example 2-4. Invoking the Librarian in Interactive Mode

```
libppc
```

The librarian prompts you for commands (libppc>). If an illegal command is entered, the librarian displays an error message and provides an opportunity to reenter the command. In this interactive mode, few librarian command errors are fatal.

Command Line Mode

You can enter librarian commands on the command line.

Example 2-5. Invoking the Librarian in Command Line Mode

```
libppc -r"sym1.o,sym2.o" -a"mod1.o,mod2.o,mod3.o" -fl abc.lib
```

The **-r** (**REPLACE**) command replaces modules in **abc.lib** with modules of the same name contained in the files **sym1.o** and **sym2.o**. The **-a** (**ADDMOD**) command adds modules contained in the files **mod1.o**, **mod2.o**, and **mod3.o** to the library **abc.lib**. The contents of **abc.lib** are listed to standard output. The final library contents will then be saved to **abc.lib**, overwriting the previous contents.

Command File Mode

You can execute librarian commands from a command file in batch mode. The commands are read in the exact order in which they are specified.

Any error encountered when processing the command file is printed, and execution of the command that generated the error is not completed. After the first error is encountered, commands are read, checked for errors, and executed if possible. However, if a library file is specified, it is not generated if an error is encountered.

Example 2-6. Invoking the Librarian in Command File (Batch) Mode

```
libppc < command_file1
```

The librarian commands in **command_file1** will be run in batch mode. Refer to Chapter 14, “Librarian Commands”, for more details on the librarian commands.

Return Codes

The assembler, linker, and librarian pass a return code to the operating system upon completion of program execution. The return code indicates whether the program executed properly or not.

The return codes are:

0 Warning or No Error.

Either the program ran to completion with no user errors (No Error) or there were user-created warnings (Warning).

1 Execution Error or Fatal Error.

Either the program ran to completion with user-created assembler, linker, or librarian errors (Execution Error) or the program did not run to completion due to a system problem (Fatal Error). A Fatal Error, for example, would occur if the program was not allocated the required amount of disk space or a file read/write error occurred.

A message describing the problem will accompany the return code for a warning, execution error, or fatal error.

Chapter 3
Assembly Language

A PowerPC microprocessor executable program consists of a sequence of 32-bit binary values contained in memory. These values represent PowerPC family instructions, memory addresses, and data. It is possible to program the microprocessor by manually calculating and encoding the values that cause the microprocessor to perform the desired functions. However, this method is very slow and tedious. An assembler provides an easier method of writing programs by allowing machine instructions to be encoded symbolically with English-like mnemonics and symbols.

This chapter describes the format of assembly language source files and assembler statements.

Assembler Statements

An assembly program consists of statements written in symbolic machine language. There are four types of assembly language statements:

- Instruction statements
- Directive statements
- Macro statements
- Comment statements

The syntax for instructions, directives, and macros is as follows:

```
[label:] [operation  [ operand [, operand ] ... ] ]    [# comment]
```

where:

label The label field assigns a memory address or constant value to the symbolic name contained in the field. The label field may begin in any column. It must be terminated by a colon (**:**).

A label can be the only field in a statement. The first 8192 characters of a label are significant. Labels are case-sensitive. For more information on labels, see the section “Labels” in this chapter.

The label may be followed by an exclamation mark (**!**) if the label is used within a macro definition or repeat block. In that context, the “**!**” is replaced with **_!BCOUNT** and is uniquely identified within each invocation of the macro or iteration of the repeat block. The “**!**” character may not follow labels if the label is not within a macro or repeat block.

operation The operation field specifies an instruction mnemonic, a directive, or a macro call. If both a label and an operation field are present, this field must be separated from the label field by a colon. The field is case-sensitive for macro names and case-insensitive for directives and instruction mnemonics. If preceded with an exclamation mark (**!**), the operation field will not match a macro name. This can be useful if a macro was defined with the same name as an instruction or directive.

operand The operand field specifies an argument for the opcode, directive, or macro specified in the operation field. The operand field, if present, is separated from the operation field by one or more blanks or tabs. Multiple operands are delimited by the comma character.

comment The comment field provides a place to put a message stating the purpose of a statement or group of statements. The comment field is always optional and, if present, must be separated from the preceding field by a pound sign (**#**).

The various fields that comprise a statement are separated by one or more blanks, tabs, commas (**,**), and in some cases, a colon (**:**).

Statement Examples

The following sections show examples of the types of assembly statements.

Instruction Statement

The symbolic machine instruction is a written specification for a particular machine operation, expressed by a symbolic operation code, also called a mnemonic, and operands. Symbolic addresses can be defined by the statement as well as used for opcode operands.

Example 3-1. Instruction Statement

```
main:    ADD      r8,r7,r6    # This instruction adds...
```

main A symbol representing the memory address of the instruction. The symbol is followed by a colon (**:**) and represents the label field.

ADD A symbolic mnemonic representing the bit pattern of the **ADD** machine instruction.

r8,r7,r6 Reserved symbols representing registers named **r8**, **r7**, and **r6**. These symbols are the operands required by the machine instruction **ADD**. The register operands are separated by commas.

In the following statements, the label **main** is defined to be the address of the statement and can be used as an operand in other statements.

```
main:    ADD   r8,r7,r6       # This instruction adds the
                              # contents of register r6 to the
```

```
                                         # contents of register r7 and
                                         # stores the result in register
                                         # br8.
       BA main                           # Set the program counter to the
                                         # address of "main".
```

Directive Statement

A directive statement is a control statement to the assembler. It is not translated into a machine instruction, and its operation field always begins with a period (**.**).

Example 3-2. Directive Statement

```
       .set     ABC,0x1000        # set ABC = 0x1000
```

.set A directive that instructs the assembler to set the first operand to the value of the second.

ABC A symbolic operand required as the first operand.

0x1000 A numeric value used as the second operand. The first and second operands are separated by a comma.

Macro Statement

A macro statement is a definition of a macro or an invocation of a macro. A macro definition may define sequences of other instructions, directives, or macros. A macro call can be made many times from any part of the program. The requirements for creating macros are described in Chapter 8, “Macros”. The macro call has the same format as a directive statement, except that the operator is one of the macro commands.

Example 3-3. Macro Statement

```
       .MACRO    M1,X,Y           # A macro definition for M1
```

.MACRO The macro definition keyword is a directive that assigns a name and a set of formal parameters. This directive also indicates that all of the following assembler statements, up to the occurrence of the **.mend** directive, are part of the macro definition.

M1 Symbolic name of the macro which is used to call the macro from other parts of the program. This name is case-sensitive.

X,Y Symbolic operands separated by commas. They represent the formal parameters of the macro and are case-sensitive.

Comment Statement

A comment statement is not processed by the assembler. Instead, it is reproduced in the assembly listing and can be used to document groups of assembly language statements. A comment statement has a pound sign as the first nonblank character on a line. All the characters that follow the pound sign (#), up to the end-of-line, are part of the comment:

```
# This is a comment statement.
```

Blank lines are also treated as comment statements.

Symbolic Addressing

When writing statements in assembly language, the machine code is usually expressed symbolically. In the following code, the machine instruction adds the contents of one source register to the contents of another source register and places the results in the destination register:

```
ADD destination_c,source_a,source_b
```

You can also attach a label to an instruction to refer to the location of the instruction symbolically, as shown in the following code:

```
label_1:    ADD destination_c,source_a,source_b
```

Whenever there is a valid identifier in the label field, the assembler assigns the address contained in the assembly program counter to the identifier. The assembly program counter is an internal variable maintained by the assembler that is set to the address of the byte currently being assembled. See the section “Assembly Program Counter” in this chapter for more information.

The identifier can then be used anywhere in the source program to refer to the instruction location. The important concept is that the identifier acts as storage for the address of the instruction location. The identifier name can then be used elsewhere in the program. When the program is translated into machine code, the address of the identifier will be inserted wherever the identifier is referenced:

```
b1:       instruction 1
              .
              .
          instruction 2

          BA b1
```

The address represented by **b1** is inserted into the **BA** instruction. That address represents the address of *instruction 1*.

Under certain circumstances, the label may not be given the offset value that might be expected. If the label is on the same line as a data directive and if the current program counter is not aligned properly for the data directive, then the program counter may be aligned to an appropriate boundary. If this happens, the label will be given the aligned value, not the original value of the program counter:

```
         .byte 1
   x:    .long 2   # x is aligned, if .alon set
```

The alignment only occurs if the automatic data alignment is turned on (using the **.alon** directive). If desired, this behavior can be turned off using the **.aloff** directive. The address of labels on lines without any directives will not be affected by the automatic alignment of data on following lines:

```
         .byte 1
      x:       # x is not aligned, whether .alon
               # set or not
         .long 2
```

Assembly Time Relative Addressing

Symbolic addresses can be used to refer to nearby locations without defining new labels. This form of addressing is called relative addressing and can be accomplished through the use of the plus (+) and minus (-) operators.

The following instruction sequences are equivalent:

Sequence 1:

```
      lab1:    ADD      r8,r7,r6
               BGT      lab2
      NOP
               B        lab1
      NOP
      lab2:    ADD      r8,r7,r6
```

Sequence 2:

```
               ADD      r8,r7,r6
      lab1:    BGT      lab1+16     # lab1+16 is the location of the 2nd ADD
                                    # instruction
               NOP
               B        lab1-4      # lab1-4 is the location of the 1st ADD
                                    # instruction
               NOP
               ADD      r8,r7,r6
```

Sequence 3:

```
               ADD      r8,r7,r6
               BGT      $+16        # Current location + 16
               NOP
               B        $-12        # Current location - 12
               NOP
               ADD      r8,r7,r6
```

The assembler will issue a warning message if the relative address generated through these facilities is not on a four-byte boundary (word-aligned). Special care should be exercised when using assembly time relative addressing, because this method is prone to errors.

Assembler Syntax

Assembly language, like other programming languages, has a character set, a vocabulary, grammar rules, and allows for the definition of new words or elements. The rules that describe the language are referred to as the “syntax” of the language.

Character Set

The assembler recognizes the alphabetical characters **A–Z** and **a–z,** the numeric characters **0–9**, and the following special characters:

&	ampersand	%	percent sign
*	asterisk	.	period
\	backslash	+	plus sign
	blank	}	right brace
:	colon	]	right bracket
,	comma	)	right parenthesis
$	dollar	;	semicolon
“	double quote	#	pound sign
=	equal sign	‘	single quote
!	exclamation mark	/	slash
>	greater than		tab
{	left brace	~	tilde
[	left bracket	_	underscore
(	left parenthesis	^	up arrow (caret)
<	less than	|	vertical bar
-	minus sign		

Any other characters, except those in a comment field or a string, generate an error. Many of the special characters have no previously defined meanings except as character constants.

Symbols

Symbols are used to represent arithmetic values, memory addresses, bit arrays (masks), and so forth. A symbol is a sequence of characters. The first 8192 characters of a symbol are significant. The first character in a symbol must be alphabetic or a special character such as an underscore (_) or a period (**.**). Subsequent characters in the symbol can consist of the above two special characters, alphabetic letters, or numeric digits. Embedded blanks are not permitted in symbols.

Example 3-4. Valid and Invalid symbols

Valid Symbols:

```
LAB1
mask
LOOP.NUM
LOOP_COUNT
L23456789012345678901234567890123456789
```

Invalid symbols:

```
ABORT*
1LAR
PAN N
```

ABORT* contains a special character. **1LAR** begins with a number. **PAN N** contains an embedded space.

Reserved Symbols

The ASMPPC Assembler contains several reserved symbols or keywords that you cannot redefine. The symbolic register names used to denote the various hardware registers are reserved symbols. Note that register names are case-insensitive. Many reserved register names are listed in Table 3-1. A complete list of register names accepted for a given PowerPC microprocessor can be obtained by using the **-fv** command line option.

Register names can be used as operands to instructions wherever a register is indicated. A constant expression can also be used to represent a register (values from 0 to 31 for general or floating-point registers, or 0 to 7 for condition registers). Either method is valid, although the **-fnr** option can be used to disallow the use of register names.

General Purpose Registers

These registers are available for most processors. The floating-point registers are available if a processor has hardware support for floating-point or if the **/F** modifier is used when specifying the processor with the **-p** option or the **.cputype** directive. The segment registers are only available on some processors.

Table 3-1. General Purpose Register Names

Names	Meaning
r0 to r31	General integer registers
sp	Stack register (r1)

Table 3-1. General Purpose Register Names (cont.)

Names	Meaning
rsda	Small data area (r13)
rsda2	Small data area 2 (r2)
rsda0	Small data area (r0)
f0 to f31	Floating-point registers
cr0 to cr7	Condition register fields
sr0 to sr15	Segment registers

Special Purpose Registers (SPR)

Special purpose register symbols denote processor registers. These symbols are only valid for use with the **mfspr**, **mftb**, or **mtspr** instructions.

The list of available SPR registers for a given PowerPC microprocessor is available in the manufacturer's User Manual. A list of accepted registers for a given PowerPC microprocessor can be obtained with the **-fv** command line option. Any register that is read-only should not be used with the **mtspr** instruction. Conversely, any register that is write-only should not be used with the **mfspr** instruction. SPR registers can be accessed either symbolically or through their equivalent SPR values.

Device Control Registers (DCR)

The 4xx series of processors can also use device control register symbols to denote processor registers. These symbols are valid only for use with the **MFDCR** and **MTDCR** instructions. The list of available DCR registers for a given PowerPC microprocessor is available in the manufacturer's User Manual. A list of accepted registers for a given PowerPC microprocessor can be obtained with the **-fv** command line option. Any register that is read-only should not be used with the **MTDCR** instruction. A register that is write-only should not be used with the **MFDCR** instruction. DCR registers can be accessed either symbolically or through their equivalent DCR values.

Altivec Vector Registers

Altivec Vector Registers are vector registers defined by the Altivec programming model. They are only available on processors that implement Altivec technology. Currently, these registers are only in the **74***xx* and 86xx series. Altivec Vector Registers are user model (UISA) registers that are used as source and destination operands for Altivec load, store, and computational instructions. Each vector register is 128 bits wide and can hold 16 8-bit elements, 8 16-bit elements, or 4 32-bit elements. The names of the Altivec Vector Registers are shown in Table 3-2.

Table 3-2. Altivec Vector Register Names

Name	Meaning
V0 to V31	Altivec Vector Registers

Performance Monitor Registers (PMR)

The **85***xx* series of processors, as well as some 51xx and 83xx variants, define a set of registers used exclusively by the performance monitor. PMRs are accessed through the **MTPMR** and **MFPMR** instructions. The list of available PMR registers for a given PowerPC microprocessor is available in the manufacturer's User Manual. Any register that is read-only should not be used with the **MTPMR** instruction. PMR registers can be accessed either symbolically or through their equivalent PMR values.

Predefined Assembler Symbols

Predefined assembler symbols can access various internal run-time values in the assembler. These symbols represent numeric values or strings that are used during the processing of the source file. Many of these values can be directly modified by directives such as **.alon**, **.pagelen**, **.title**, and **.maxdepth**. These symbols allow access to, but not modification of, these values for conditional tests, use as data, or other reasons. These symbols are accessed by using an exclamation mark (**!**) followed immediately by the name of the assembler symbol. All assembler symbols are case-sensitive, so the exact form of the name must be used. The assembler symbol may be used anywhere that a similar value or string is valid.

Table 3-3. Predefined Assembler Symbols

Assembler Symbol	Type	Meaning
ALDATA	Number	If automatic alignment is turned on, 1; otherwise, 0.
BCOUNT	Number	Value of MCOUNT at the beginning of the current macro or repeat block expansion.
LPAGE	Number	Maximum line count for a listing page.
LSTITLE	String	Current subtitle for listing.
LTITLE	String	Current title for listing.
LWIDTH	Number	Maximum width of a listing line.
LWRAP	Number	If line wrapping is turned on, 1; otherwise, 0.
MAXDEPTH	Number	Maximum macro recursion depth.
MCOUNT	Number	Cumulative count of macro or repeat block expansions.

Table 3-3. Predefined Assembler Symbols (cont.)

Assembler Symbol	Type	Meaning
MDEPTH	Number	Current recursion level of macro or repeat block.
MEXP	Number	If macro expansion in listing is turned on, 1; otherwise, 0.
MLIST	Number	Nonblank argument count for macro (0 outside of macros).

Example 3-5. Using Predefined Assembler Symbols

```
.byte !ALDATA   # value of data alignment setting
.string !LTITLE # listing title
# perform alignment if automatic alignment is
# turned off
.if !! !ALDATA
.align 4
.endif
```

Relocatable Symbols

Each ASMPPC symbol has an associated symbol type that denotes the symbol as absolute or relocatable. If relocatable, the type also indicates the section where the symbol resides (such as **.text** or **.data**). Symbols whose values are not dependent on program origin are called absolute symbols. Symbols whose values change when the program origin is changed are called relocatable symbols.

All external references are considered relocatable even though the external can be defined as an absolute value at link time. When assembling a module, the assembler does not know whether an externally defined symbol in another module is relocatable or absolute and, therefore, assumes the external is relocatable.

Absolute and relocatable symbols can appear in relocatable program sections. The characteristics of absolute and relocatable symbols are as follows:

- A symbol is absolute if it:
    - Is defined to be a constant expression
    - Is defined (as labels) in dummy sections
- A symbol is relocatable if it:
    - Appears in the label field of an instruction that is in a relocatable section
    - Has been declared by the **.extern**/**.globl** directive
    - Refers to the program counter (**$**) while assembling a relocatable section

Example 3-6. Absolute and Relocatable Symbols

```
    .sect newdata
    .long 0xABCDEF
    .sect .data# the first appearance of this
# section in this file
L1: .long 0xABCD# L1 is a relocatable symbol
    .sect .newdata
    .byte 80
    .sect .data # the second appearance of this
# section in this file
L2: .short 700# L2 is a relocatable symbol
```

Assembly Program Counter

During the assembly process, the assembler maintains a variable that always contains the address of the current assembly location being assembled. This variable is called the assembly program counter. It is used by the assembler to assign addresses to assembled bytes, but it is also available to the programmer. The dollar sign (**$**) is the symbolic name of the assembly program counter. It can be used like any other symbol, but it cannot appear in the label field. Any attempt to use the assembly program counter to branch to an instruction that is not word-aligned (for example, **BA $+ 2**) will produce an error at link time.

Example 3-7. Using the Assembly Program Counter

If the following instruction is located at address 0x10:

```
B $
```

then the instruction would actually be assembled as:

```
B 0x10
```

The relative branch instruction is at location 0x10. The instruction directs the microprocessor to branch to the beginning of the current instruction (location 0x10). The program counter in this example contains the value 0x10 and the instruction will be translated to a relative jump to location 0x10 from location 0x10 (an infinite loop). This example is useful when waiting for an interrupt.

Other Reserved Symbols

Some other predefined symbols exist within the assembler. The section names in Table 3-4 are always defined and have certain predefined attributes (see the **.sect** directive in Chapter 8, “Macros”). These names are unavailable for redefinition by the user. Some special operators exist, but these symbols are recognized depending upon their context. You can define new symbols with the same names as these operators.

Table 3-4. Other Reserved Symbols

Category	Symbol
Assembly Program Counter	$
Predefined Section Names	.bss
	.data
	.data1
	.init
	.fini
	.PPC.EMB.sbss0
	.PPC.EMB.sdata0
	.rodata
	.rodata1
	.sbss
	.sbss2
	.sdata
	.sdata2
	.text
Expression Operators	hi
	ha
	lo
	sda
	sdaaddr
	secthi
	sdabase
	sdaoff
	sectha
	sectlo
	sectoff
	sectof
	sda2addr

Table 3-4. Other Reserved Symbols (cont.)

Category	Symbol
Expression Operators (cont.)	sda2base
	sizeof
	sizeha
	sizehi
	sizelo
	startha
	starthi
	startlo
	startof

Constants

A constant is an invariant quantity. It can be an arithmetic value or a character code. Arithmetic values can be represented in either integer or floating-point format.

Integer Constants

In most cases, integer constants must be contained in one, two, or four bytes. An equivalent two's complement representation is generated for a negative constant and placed in the field specified. The ranges of values for constants are shown in Table 3-5.

Table 3-5. Constant Value Ranges

Number of Bytes	Value Range
1 (unsigned)	0 to 255
2 (unsigned)	0 to 65,535
4 (unsigned)	0 to 4,294,967,295
1 (two's complement)	-128 to +127
2 (two's complement)	-32,768 to +32,767
4 (two's complement)	-2,147,483,648 to +2,147,483,647

Numbers whose most significant bit is set can be interpreted either as a large positive number or a negative number. The assembler correctly recognizes numbers in either form; however, you are generally responsible for their interpretation.

All constants are evaluated as 32-bit quantities (that is, modulo 2^{32}). Whenever an attempt is made to place a constant in a field for which it is too large, the assembler generates a warning message.

Decimal constants are a sequence of numeric characters optionally preceded by a plus or a minus sign. If unsigned, the value is assumed to be positive. Constants with bases other than decimal are defined by specifying a coded descriptor or special character before the constant. Table 3-6 shows the available prefixes and their meanings. If no prefix is given, the default number base (which is decimal) is used.

Table 3-6. Prefixes Used Before Constants

Prefix	Constant Base
0b or 0B	Binary
no prefix	Decimal (default)
0o or 0O	Octal
0x or 0X	Hexadecimal

Example 3-8. Using Prefixes for Constants

0b10 or **0B10**	Specifies a value of 2 (binary base)
0o10	Specifies a value of 8 (octal base)
10	Specifies a value of 10 (decimal base)
0x10 or **0X10**	Specifies a value of 16 (hexadecimal base)

Floating-Point Constants

A floating-point constant is a string of characters and digits that represent an IEEE-formatted floating-point number. In the following syntax, the character **E** is used to specify constants in scientific notation; it can be uppercase or lowercase:

```
[{+|-}]dec[.[dec]][e|E[{+|-}]dec]
```

where *dec* represents a decimal integer.

Example 3-9. Floating-Point Constants

`-1.e-3`	(Same as -0.001)
`3.14159`	(PI to 5 places)
`+3.333E+5`	(Same as 333300.0)
`0`	(Zero)
`0.4`	(Zero required before decimal point)

Floating-point constants can only appear in the data storage directives **.double**, **.uadouble**, **.float**, **.uafloat**, and **.set**. Decimal constants can appear anywhere that floating-point constants are required but will be converted to floating-point value by the assembler.

If a floating-point constant does not have a decimal point, the decimal pointer is assumed to be to the right of the least significant digit. Also note that at least one digit must appear before the decimal point, otherwise the constant will be interpreted as an identifier.

Character and String Constants

Characters can also be used as constants. A single character placed after a single quote is an integer constant:

'a (Equivalent to the constant 0x61)

Strings can be used as constants to supply a sequence of values for the **.string**, **.error**, **.file**, **.byte**, **.ifeqs**, and **.ifnes**. directives. A string constant is a sequence of characters enclosed in double quotes (**" "**). The largest string that can be represented is 8200 characters in length (including any terminating NULL character, if one is emitted). All the directives except **.byte** emit a terminating NULL character:

```
"This is an example of a string constant."
```

Character and string constants can contain any character. The backslash (\) is used within character and string constants to escape the quote marks and to specify control characters (see Table 3-7). If the character following a backslash is not one of those specified in the table, the backslash is ignored. Note that the backslash is also not recognized as an escape character in directives that specify filenames, such as the **.file** and **.include** directives.

Table 3-7. Summary of Control Characters

Characters	Meaning
\\	Backslash character
\"	Double quote character
\f	Form feed
\n	Newline character

Table 3-7. Summary of Control Characters (cont.)

Characters	Meaning
\0	NULL character
\r	Return character
\’	Single quote character
\t	Tab character
ddd	Octal digits specifying the value of the desired character
\x*hhh*	Hex digits (3 maximum) specifying the value of the desired character up to 0x0ff; the x must be lowercase

Labels

Labels are used to mark instruction and data storage locations. Normal labels are identifiers that can begin in any column and must be followed by a colon. Local labels consist of a single decimal digit followed by a colon. Local labels are described below. The label is given a value that represents the current program counter within the current section. If the current section is a dummy section, the label's value simply represents an absolute offset from the start of the dummy section. The value that is assigned may be affected by any realignment that the assembler has to do for any data directive that appears on the same line as the label.

Normal labels can be defined only once and are visible throughout the entire module. To make a normal label visible to another assembly module, it must be used in a **.globl** or **.extern** directive. The module referencing the global label must use the **.globl** or **.extern** directive as well.

Assembler-modified labels can be defined many times.

Example 3-10. Labels

```
Lab!:
```

If this label were used in a macro definition, the first invocation of the macro would create the label **Lab_1** (provided this was the first invocation in the source file). The next invocation would create the label **Lab_2**.

- Assembler-modified labels are nested within macro calls.
- Assembler-modified labels are nested within **.rept**, **.irep**, and **.irepc** directives.

Assembler-modified labels are useful in macro declarations where the macro can be invoked many times.

Local labels consist of a single decimal digit (for example, 1). As with normal labels, local labels can begin in any column and must be followed by a colon. A local label can occur multiple times within a file. A reference to a local label is made by appending a ‘b’ (backward reference) or ‘f’ (forward reference) to the digit. The assembler will associate the closest preceding (‘b’) or following (‘f’) local label to the reference. The reference must occur in the same section as the label. Local labels can be used in C/C++ asm() statements to avoid ASMPPC’s duplicate symbol errors due to inlining or macro inclusions. Local labels are like normal labels except in the following respects:

- Local labels cannot be made **.globl** or **.extern**.
- The same local label can be defined many times within a file.
- Local labels do not appear in the symbol table or in cross reference listings.
- Local label references must be made within the same section as the local label and are made by appending the label with an ‘f’ or ‘b’ to indicate a forward or backward reference.

Example 3-11. Local Labels

```
     b       1f
0:   addi    r3,r3,1
1:   cmpwi   r3,10
     bne     0b
8:  .long    8b        # self-address
```

Expressions

An expression is a sequence of one or more symbols, constants, or other syntactic structures separated by arithmetic operators. An expression is evaluated modulo 2^{32} and must resolve to a single unique value that can be contained in 32 bits. Whenever an attempt is made to place an expression in a one- or two-byte field and the calculated result is too large to fit, a warning message is generated. All expressions are considered to be signed 32-bit values, but a special case occurs when the expression is either a single constant value with no other operators, or if the expression is the result of a **LO**, **HI**, or **HA** operator. Under these conditions, if all of the high-order 16-bits of the expression are all 0's or 1's, the value will be considered to fit within a 2-byte field. However, it should be noted that some assembly instructions perform sign extension on their operands, so care should be taken when using constants with these instructions.

Example 3-12. Valid Expressions

```
.if  val<0x10000
LWZ  r3,val(0)
.else
ADDIS r3, 0, val >> 16
ORI  r3,r3,val & 0xffff
.endif
```

The comparison operators return 1 if the comparison is true and zero if the comparison is not true. Expressions with comparison operators are typically used with the **.if** directive. **val** is a symbol defined previously in the program.

Embedded blanks or tabs are allowed in expressions. Table 3-8 specifies the list of operators arranged in descending order of precedence. Operators within the same block (designated by double lines) have the same level of precedence and apply from left to right in an expression. OPR is defined to be an expression.

Table 3-8. Summary of Operators and Their Precedences

Operator	**Meaning**
(OPR)	**OPR's value**
ha (OPR)	See Table 3-9
hi (OPR)	See Table 3-9
lo (OPR)	See Table 3-9
sda (OPR)	See Table 3-9
sdaaddr (OPR)	See Table 3-9
sdaoff (OPR)	See Table 3-9
sdabase (OPR)	See Table 3-9
sda2addr (OPR)	See Table 3-9
sda2base (OPR)	See Table 3-9
sectoff (OPR)	See Table 3-9
sectof (OPR)	See Table 3-9
sectha (OPR)	See Table 3-9
secthi (OPR)	See Table 3-9
sectlo (OPR)	See Table 3-9
sizeof (OPR)	See Table 3-9
sizeha (OPR)	See Table 3-9
sizehi (OPR)	See Table 3-9
sizelo (OPR)	See Table 3-9
startha (OPR)	See Table 3-9
starthi (OPR)	See Table 3-9
startlo (OPR)	See Table 3-9
startof (OPR)	See Table 3-9

Table 3-8. Summary of Operators and Their Precedences (cont.)

Operator	Meaning
~	One’s complement
+	Unary plus
-	Unary minus
%	Modulus
*	Multiplication
/	Division
<<	Shift left
>>	Shift right
&	Bitwise AND
^	Bitwise exclusive OR
|	Bitwise inclusive OR
+	Addition
-	Subtraction
<	Less than
<=	Less or equal
>	Greater than
>=	Greater or equal
==	Equal
!=	Not equal
!!	Logical NOT
&&	Logical AND
||	Logical OR

All operations are performed using signed arithmetic. The comparison operators (<, <=, ==, **!**=, >=, >) assume the values to be compared are signed. They evaluate to true (1) if the comparison succeeds or false (0) if the comparison fails.

Expression Operators

The following table describes unary expression operators.

Table 3-9. Special Expression Operators

Operator	Description
ha()	High adjusted 16 bits of operand.
hi()	High 16 bits of operand.
lo()	Low 16 bits of operand.
sda()	Offset of operand from the base of the small data area containing the symbol in the operand. The operand must be relocatable. The base register is modified appropriately.
sdaaddr()	Offset of **.sdata** entry from the value of symbol **_SDA_BASE_** that contains the full 32-bit address of the operand. The operand is relocatable.
sdaoff()	Offset of operand from the base of the small data area containing the symbol in the operand. The operand is relocatable.
sdabase()	Offset of the operand from the value of the symbol **_SDA_BASE_**. The operand is relocatable.
sda2addr()	Offset of **.sdata2** entry from the value of symbol **_SDA2_BASE_** that contains the full 32-bit address of the operand. The operand is relocatable.
sda2base()	Offset of the operand from the value of the symbol **_SDA2_BASE_**. The operand is relocatable.
sectoff()	The operand’s value is replaced by the symbol’s address minus the start of the section containing that symbol. The operand is relocatable.
sectof()	Full 32 bits of the address of the beginning of the section containing the symbol in the operand. The operand is relocatable.

Table 3-9. Special Expression Operators (cont.)

Operator	**Description**
sectha()	High adjusted 16 bits of the address of the beginning of the section containing the symbol in the operand. The operand is relocatable.
secthi()	High 16 bits of the address of the beginning of the section containing the symbol in the operand. The operand is relocatable.
sectlo()	Low 16 bits of the address of the beginning of the section containing the symbol in the operand. The operand is relocatable.
sizeof()	Calculates the size of the specified section.
sizeha()	Calculates the high adjusted 16 bits of the size of the specified section.
sizehi()	Calculates the high 16 bits of the size of the specified section.
sizelo()	Calculates the low 16 bits of the size of the specified section.
startof()	The starting address of the specified section.
startha()	High adjusted 16 bits of the beginning address of the specified section.
starthi()	High 16 bits of the beginning address of the specified section.
startlo()	Low 16 bits of the beginning address of the specified section.

LO — Extracts Lower 16 Bits

HI — Extracts Upper 16 Bits

HA — Extracts Upper 16 Bits and Adjusts

Syntax

lo (*OPR*)

hi (*OPR*)

ha (*OPR*)

Description

- **OPR**

  A relocatable or absolute integer operand

The **lo()** operator yields the low 16 bits of the operand.

The **hi()** operator yields the high 16 bits of the operand.

The **ha()** operator yields the high-adjusted 16 bits of the operand.

These operators extract a 16-bit quantity from a 32-bit integer quantity. The operand can be absolute or relocatable. The result can be used for any signed or unsigned 16-bit operand, or anywhere that a 16-bit value can be stored.

When *OPR* is an absolute integer, the assembler will perform the operations at assembly time. The calculations the assembler performs have the following functionality (as described in the ANSI C notation):

lo (*OPR*) = *OPR* & 0x0000FFFF

hi (*OPR*) = (*OPR* >> 16) & 0x0000FFFF

ha (*OPR*) = ((*OPR* >> 16) + (*OPR* >> 15) & 0x1)) & 0x0000FFFF

When *OPR* is a relocatable quantity, the linker, rather than the assembler, will perform the operations. In addition, the *OPR* must be an expression that conforms to the following context-free grammar:

OPR ::= VAL2 | VAL3

VAL2 ::= REL | REL + ABS | REL - ABS | ABS + REL

VAL3 ::= ABS - REL

ABS denotes an absolute symbol, constant, or expression. *REL* denotes a relocatable symbol or expression. If the expression does not conform to the above form, an error message is generated by the assembler.

The assembler emits special relocation information to the object file when these operators are used with relocatable operands. During an absolute link, the linker will read this information and perform the same calculations as shown above (that is, only if the *REL* used in the expression has been properly defined).

Notes

Since most PowerPC instructions can only hold 16-bit values, the **ha()**, **hi()**, and **lo()** operators are useful in breaking 32-bit addresses into two related 16-bit values. The **lo()** operator always returns the low-order 16-bits of an address. If that operand is used in an instruction that performs sign-extension, the register that is used to hold the high-order bits of the expression must be loaded with a value that is created using the **ha()** operator. This is because the sign-extension performed on the low-order bits can result in the high-order bits being affected if the left-most bit of the low-order value is 1. The calculation of the **ha()** operator takes this sign-extension into account and adjusts for it. If the low-order 16-bits of an address are used with an instruction that does not perform sign-extension, then the **hi()** operator is the appropriate operator to use for loading a register with the high-order bits. See Example 3 for an example of the use of the **ha()** operator.

When used with the **.long** data directive, the result is placed into either the lower addressed 16 bits or upper addressed 16 bits of the 32-bit field, depending upon the byte order setting. See Chapter 6, "Assembler Directives", and the **.endian** directive for a further description of byte order.

See Chapter 5, "Relocation", for further description of relocatable expressions.

Example 1

The following example illustrates how to extract 16 bits from an absolute integer operand. The result is stored in a 16-bit field by means of the **.short** data directive.

```
Command line: asmppc -l lohiha16.s
Line  Address   Opcode        Source
1                              .sect lohiha16
2     00000000 12 34           .short lo (0xabcd1234)
3     00000002 AB CD           .short hi (0xabcd1234)
4     00000004 00 01           .short ha (0x00008000)
```

The **Opcode** column in this listing contains the results of the assembly time calculations for these operators. The result of the **ha()** operator:

```
HA (0x00008000)
   =((0x00008000 >> 16) + (0x00008000 >> 15) & 0x1)) & 0x0000FFFF
   =(0x0000 + (0x0001 & 0x1)) & 0x0000FFFF
   = 0x0001 & 0x0000FFFF = 0x0001
```

Example 2

The following example illustrates how to extract 16 bits from an absolute integer operand. The result is stored in a 32-bit field by means of the **.long** data directive.

```
Command line: asmppc -l lohiha32.s
Line    Address   Opcode         Source
1                                 .sect lohiha32
2       00000000 00 00 12 34      .long lo (0xabcd1234)
3       00000004 00 00 AB CD      .long hi (0xabcd1234)
```

```
4       00000008 00 00 00 01     .long ha (0x00008000)
```

The **Opcode** column in this listing contains the results of the assembly time calculations. The results are placed in a 32-bit field.

Example 3

This example illustrates how to form a 32-bit address using the **ha()** and **lo()** operators. In this example, the final result is not determined until link time. The first listing shows the original assembly listing:

```
Command line: asmppc -l halo32.s
Line    Address  Opcode          Source
1                                 .sect halo32
2       00000000 3C 20 00 00   R  addis  r1, r0,HA(L01)
3       00000004 80 41 00 00   R  lwz    r2, LO(L01)(r1)
4                                L01:
```

If the section **halo32** is placed at **0x1234f000**, the final object code generated for this section, after linking, is shown in the following code segment. Locating a section at a specified address may be accomplished by using the linker **ORDER** command.

```
Line    Address  Opcode          Source
1                                 .sect halo32
2       1234f000 3C 20 12 35      addis  r1,r0,0x1235
3       1234f004 80 41 f0 08      lwz    r2,0xf008(r1)
4                                L01:
```

For purposes of illustration, the object code generated is shown in terms of an assembly listing. Normally, the object code is contained in the resultant absolute file produced by the linker. For simplicity, no other module sections of the same name have been combined.

The first instruction operand on line 2, **addis HA(L01)**, is left-shifted by 16 (a function of the **addis** instruction) and stored in register **r1**.

```
HA (L01) = HA (0x1234f008)
=((0x1234f008 >> 16) + (0x1234f008 >> 15) & 0x1)) & 0x0000FFFF
=(0x1234 + (0x2469 & 0x1)) & 0x0000FFFF
=(0x1234 + 0x1) & 0x0000FFFF
=0x1235
```

The final result placed into **r1** is **0x12350000**.

The second instruction on line 3, **lwz**, adds the contents of **r1** to **LO(L01)** and, using that value as an address, loads the referenced four bytes into **r2**. **r1** contains **0x12350000**. **LO(L01)** is **0xF008**, which is sign-extended to **0xFFFFF008**. The addition of **r1** and **LO(L01)** is: **0x12350000 + 0xFFFFF008 = 0x1234F008**, which is the final address of the label **L01**.

SECTOFF — Offset Relative to Section Address

SECTOF — Full 32 bits of Section Address

SECTLO — Low 16 bits of Section Address

SECTHI — High 16 Bits of Section Address

SECTHA — High Adjusted 16 Bits of Section Address

Syntax

sectoff (*OPR*)

sectof (*OPR*)

sectlo (*OPR*)

secthi (*OPR*)

sectha (*OPR*)

Description

- **OPR**

  A relocatable operand

The **sectoff()** operator yields the signed 16-bit offset from the start of the section containing the relocatable symbol in *OPR*. The linker performs a range check to validate whether the offset fits within the 16-bit field. If not, a link-time error is produced.

The **sectof()** operator yields the full 32 bits of the start address of the section containing the relocatable symbol in *OPR*.

The **sectlo()** operator yields the low 16 bits of the start address of the section containing the relocatable symbol in *OPR*. This is the same as applying the **lo()** operator to the section base address.

The **secthi()** operator gives the high 16 bits of the start address of the section containing the relocatable symbol in *OPR*. This is the same as applying the **hi()** operator to the section base address.

The **sectha()** operator gives the high adjusted 16 bits of the start address of the section containing the relocatable symbol in *OPR*. This is the same as applying the **ha()** operator to the section base address.

These operators (aside from **sectof()**) have the effect of extracting a 16-bit quantity from a 32-bit quantity. The *OPR* must be a relocatable expression that conforms to the following context-free grammar:

OPR	::= *VAL2*
VAL2	::= *REL* | *REL* + *ABS* | *REL* - *ABS* | *ABS* + *REL*

ABS is an absolute symbol, constant, or expression. *REL* is a relocatable symbol or expression. If the expression does not conform to the above form, an error message is generated by the assembler.

If *OPR* is valid, the assembler emits special relocation information to the object file when these operators are used. During an absolute link, the linker will read this information and perform the final operations (only if the *REL* used in the expression has been properly defined).

The start address of the section is the memory location of the first byte of the combined section that is created at absolute link time. It is not necessarily the same as the address of the first byte of the section in this module.

Notes

When **sectoff()**, **sectlo()**, **secthi()**, and **sectha()** are used with the **.long** data directive, the result is placed into either the lower 16 bits or upper 16 bits of the 32-bit field, depending on the byte order setting. See Chapter 6, “Assembler Directives”, and the **.endian** directive for further description of byte order. None of these operators can be used as the target of another operator except for the **sectof()** operator, which can be used as the target operator of the **ha()**, **hi()**, and **lo()** operators.

Example 1

The following example illustrates the results of using these operators. For purposes of illustration, data directives are used. The operand used for these operators is a label. The results are placed into 16-bit (**.short**) and 32-bit (**.long**) fields. The assembler reports an error if sectof() is used for 16-bit fields, such as with the **.short** directive. The first listing is the original assembler listing.

```
Command line: asmppc -l sectops.s
Line   Address  Opcode         Source
1                              .sect sectops
2     00000000 00 08        R .short sectoff (here)
3     00000002 00 00       R .short sectlo (here)
4     00000004 00 00        R .short secthi (here)
5     00000006 00 00        R .short sectha (here)
6                              here:
7     00000008 00 00 00 08  R .long  sectoff (here)
8     0000000C 00 00 00 00  R .long  sectof (here)
9     00000010 00 00 00 00  R .long  sectlo (here)
10    00000014 00 00 00 00  R .long  secthi (here)
11    00000018 00 00 00 00  R .long  sectha (here)
```

The actual results are produced by the linker. For purposes of illustration, the object code generated is shown in terms of an assembly listing. Normally, the object code is contained in the resultant absolute file produced by the linker. In this example, no other module sections of the same name have been combined.

Supposing the start address of the section **sectops** is **0x00018000**, the final object code in **sectops** would be:

```
Line   Address  Opcode         Source
1                              .sect sectops
```

```
2       00000000 00 08          .short 0x0008
3       00000002 80 00          .short 0x8000
4       00000004 00 01          .short 0x8000
5       00000006 00 02          .short 0x0001
6                               here:
7       00000008 00 00 00 08    .long  0x00000008
8       0000000C 00 01 80 00    .long  0x00018000
9       00000010 00 00 80 00    .long  0x00008000
10      00000014 00 00 00 01    .long  0x00000001
11      00000014 00 00 00 02    .long  0x00000002
```

The result of the **sectoff()** operator is the offset of the label **here** from the beginning of the section **0x00018008 - 0x00018000 = 0x0008**.

The **sectof()** operator gives a 32-bit section base address for the **.long** data directive, which, in this case, is **0x00018000**.

The result of the **sectlo()** operator applies the **lo()** operator to the base address of the section **sectops**. The base address is **0x00018000** and the lower 16 bits are **0x8000**.

The result of the **secthi()** operator applies the **hi()** operator to the base address of the section. The base address is **0x00018000** and the upper 16 bits are **0x0001**.

The result of the **sectha()** operator applies the **ha()** operator to the base address of the section. The calculation is:

```
ha (0x00018000)
= ((0x00018000 >> 16) + (0x00018000 >> 15) & 0x0001)) & 0xFFFF
= (0x0001 + 0x0003 & 0x0001) & 0xFFFF
= (0x0001 + 0x0001) & 0xFFFF
= 0x0002 & 0xFFFF
= 0x0002
```

The results of these operators are then placed into the **.short** and the **.long** fields where they are used. The **Opcode** column of the second listing shows the results.

Example 2

The next example shows how to load the start address of a section into a register. Suppose there is a label named **label** in a section named **data**. Using this label with the **secthi()** and **sectlo()** operators will work. The following listing is the original assembler listing.

```
Command line: asmppc -l startof.s
Line    Address  Opcode         Source
1                               .sect startof
2       00000000 3C 60 00 00  R addis r3,r0,secthi(label)
3       00000004 60 63 00 00  R ori r3,r3,sectlo(label)
4                               .sect data
5                               label:
```

Suppose the start address of the **data** section is **0x00048000**. The final object code for the **startof** section after linking would be:

```
Line    Address  Opcode         Source
1                               .sect startof
2       00000000 3C 60 00 04    addis r3,r0,0x0004
```

```
3      00000004 60 63 80 00    ori r3,r3,0x8000
```

The linker takes **HI(0x00048000)**and **LO(0x00048000)** and stores the results in 16-bit displacement fields of the **addis** and **ori** instructions.

Example 3

As in **Example 2**, this example calculates the address of an external symbol and then loads the value at that address into a register. If there are the following two assembly modules:

```
Command line: asmppc -l effaddr.s
Line   Address  Opcode         Source
1                              .extern esym
2                              .sect mysect
3      00000000 3C 80 00 00  E  addis r4,r0,sectha(esym)
4      00000004 38 84 00 00  E  addi  r4,r4,sectlo(esym)
5      00000008 80 A4 00 00  E  lwz   r5,
                     sectoff(esym)(r4)

Command line: asmppc -l symsect.s
Line   Address  Opcode         Source
1                               .globl esym
2                               .sect symsect
3      00000000 01 23 45 67     .long 0x01234567
4      00000004                esym:
5      00000004 89 AB CD EF     .long 0x89abcdef
```

then after final linkage of **effaddr.o** and **symsect.o**, where the section **symsect** is located at **0x00048000** and the location of **mysect** is at zero, the resultant object code in the section **mysect** would be:

```
Line   Address  Opcode         Source
1                              .extern esym
2                              .sect mysect
3      00000000 3C 80 00 05     addis r4,r0, 0x0005
4      00000004 38 84 80 00     addi  r4,r4, 0x8000
5      00000008 80 A4 00 04     lwz   r5, 0x4 (r4)
```

On line 3, the linker applies the **ha()** operator to the base address of **symsect**, **0x00048000**, because **esym** is contained within this section. Thus, **0x0005** is stored in the 16-bit signed immediate field of the **addis** instruction.

Similarly, on line 4 the linker applies the **lo()** operator to the base address of **symsect**, storing the value **0x8000** into the **addi's** signed immediate field.

On line 5, the linker determines the offset of **esym** from the start address of the section **symsect**,**0x00048004 - 0x00048000 = 0x4**, and stores this value in the displacement field of the **lwz** instruction.

This combination of instructions and use of the special PowerPC relocation operators makes it easier to calculate effective addresses.

Example 4

This example shows how to load the start address of a section into a register, as in Example 2, but, in this case, through the use of the predefined **LOAD** macro. For more details on **LOAD** and **STORE** macros, refer to the section Predefined Macros in Chapter 8, Macros.

Suppose there is a section named **data**. Using this section with the **sectof()** operator works in conjunction with the **LOAD_ADDR** macro. The following is the original assembler listing:

```
Command line: asmppc -l -fm sectof.s
Line  Address     Opcode           Source
1                                  .sect startof
2                                  LD_ADDR r3, sectof(data)
2.1   00000000    3C 60 00 00  R   addis r3,0,ha(sectof(data))
2.2   00000004    30 63 00 00  R   addic r3,r3,lo(sectof(data))
3                                  .sect data
```

Suppose the start address of the data section is **0x00048000**. The final object code for the **startof** section after linking would be as follows:

```
Command line: asmppc -l -fm sectof.s
Line  Address     Opcode           Source
1                                  .sect startof
2                                  LD_ADDR r3, sectof(data)
2.1   00000000    3C 60 00 05  R   addis r3,0,ha(sectof(data))
2.2   00000004    30 63 80 00  R   addic r3,r3,lo(sectof(data))
3                                  .sect data
```

The operator combination **ha(sectof())** is the same as the **sectha()** operator, and **lo(sectof())** is the same as **sectlo()**.

SDABASE — 16-Bit Offset From _SDA_BASE_
SDA2BASE — 16-Bit Offset From _SDA2_BASE_

Syntax

sdabase (*OPR*)

sda2base (*OPR*)

Description

- **OPR**

  A relocatable integer operand

The **sdabase()** operator gives the signed 16-bit offset of the symbol in *OPR* from the symbol **_SDA_BASE_** (a special linker generated symbol).

The **sda2base()** operator gives the signed 16-bit offset of the symbol in *OPR* from the symbol **_SDA2_BASE_** (a special linker generated symbol).

These operators have the effect of extracting a 16-bit quantity from a 32-bit quantity. The *OPR* must be a relocatable expression that conforms to the following context free grammar:

OPR ::= VAL2

VAL2 ::= REL | REL + ABS | REL - ABS | ABS + REL

ABS is an absolute symbol, constant, or expression. *REL* is a relocatable symbol or expression. If the expression does not conform to the above form, an error message is generated by the assembler.

If *OPR* is valid, the assembler emits special relocation information to the object file when these operators are used. During an absolute link, the linker will read this information and perform the final operations (only if the *REL* used in the expression has been properly defined). The linker performs a range check to validate whether the offset fits within the 16-bit field. If not, a link-time error is produced.

Notes

When used with the **.long** data directive, the result is placed into either the lower 16 bits or upper 16 bits of the 32-bit field, depending upon the byte order setting. See Chapter 6, “Assembler Directives”, and the **.endian** directive for further description of byte order.

Example

This example shows how to calculate the offsets of the symbols **offsdabase** and **offsda2base** from the **_SDA_BASE_** and **_SDA2_BASE_** special linker symbols. These offsets are stored into 16-bit storage locations. The original assembler listing is as follows:

```
Command line: asmppc -l sdabase.s
Line   Address  Opcode          Source
1                               .sect .sdata
2       00000000 00 00 00 01     .long 1
```

```
3                              .sect .sdata2
4      00000000 00 00 00 02    .long 2
5                              .sect small_data_bases
6
7                              .extern offsdabase
8                              .extern offsda2base
9
10                             offsdabase:
11     00000000 00 00       R .short sdabase(offsdabase)
12                             offsda2base:
13     00000002 00 02       R .short sda2base(offsda2base)
```

Lines 11 and 13 are the points of interest. Assume that after an absolute link, the symbols had the following locations (in hexadecimal):

```
offsda2base = 0x0000000A
offsdabase  = 0x00000008
_SDA2_BASE_ = 0x00000006
_SDA_BASE_  = 0x00000002
```

The linker calculates the difference between the user-defined symbols and the special linker symbols as follows:

```
sdabase(offsdabase)
= offsdabase - _SDA_BASE_
= 0x00000008 - 0x00000002
= 0x6

sdabase(offsda2base)
= offsda2base - _SDA2_BASE_
= 0x0000000A - 0x00000006
= 0x4
```

As done previously, the final object code generated is shown in terms of the original assembly listing (only lines 10-13 are shown). The linker places these results into the two storage locations specified with the **.short** data directive:

```
10                             offsdabase:
11     00000010 00 06          .short  0x0006
12                             offsda2base:
13     00000012 00 04          .short  0x0004
```

SDA — Calculates Link Time 16-bit Offset and Register Base From Implicit Small Data Area Base

Syntax

sda (*OPR*)

Description

- **OPR**

  A relocatable integer operand

The **sda()** operator gives the signed 16-bit offset of the symbol in *OPR* from either:

- **_SDA_BASE_**, if the symbol is in **.sdata** or **.sbss**
- **_SDA2_BASE_**, if the symbol is in **.sdata2** or **.sbss2**
- Zero (0), if the symbol is in **.PPC.EMB.sdata0** or **.PPC.EMB.sbss0**

This relocation also causes the RA field of the instruction where it is used to be set to the base register of the appropriate SDA.

Thus, usage of the **sda()** operator is restricted to only those instructions having D-forms. This operator is allowed in **ADDIC**, **ADDIC.**, **ADDI**, **SUBI**, **SUBIC**, **SUBIC.**, **LA**, **LI**, and those load and store instructions with an address register and displacement. It is not allowed in any other context.

OPR must be an expression that conforms to the following context-free grammar:

```
OPR        ::= VAL2
VAL2       ::= REL | REL + ABS | REL - ABS | ABS + REL
```

ABS denotes an absolute symbol, constant, or expression. *REL* denotes a relocatable symbol or expression. If the expression does not conform to the above, an error message is generated by the assembler.

If *OPR* is valid, the assembler emits special relocatable information to the object file. During an absolute link, the linker will read this information and calculate the final address values (only if the *REL* used in the expression has been properly defined). The usage of this operator affects a 21-bit field: the 16-bit displacement and the 5-bit RA register field.

Notes

The value placed in the RA field of the instruction is determined at link time. The value represents the base register of a small data area. The linker determines the base register according to which section the symbol used in the expression is found within. If the symbol is not found in one of these sections, the link will fail.

Table 3-10. Link-Time Values for Base Register

Symbol Location	Value
.sdata .sbss	13 (for GPR13)
.sdata2 .sbss2	2 (for GPR2)
.PPC.EMB.sdata0 .PPC.EMB.sbss0	0 (for GPR0)

Example

This example illustrates how the **sda()** operator causes the linker to transform instructions when the **sda()** operator is used. The linker picks the appropriate SDA base according to the small data area the specified symbol falls in. The following listing is the original assembler listing:

```
Command line: asmppc -l sdaex.s
Line   Address  Opcode         Source
1                              .sect .sdata
2      00000000 00 00 00 01    sdaval: .long 1
3                              .sect .sdata2
4      00000000 00 00 00 02    sda2val: .long 2
5                              .sect .PPC.EMB.sdata0
6      00000000 00 00 00 03    sda0val: .long 3
7                              .sect asmsect
8      00000000 80 B2 00 00  R lwz r5,sda(sdaval)(r18)
9      00000004 38 E8 00 00  R addi r7, r8,sda(sdaval)
10     00000008 81 33 00 00  R lwz r9,sda(sda2val)(r19)
11     0000000C 81 42 00 00  R lwz r10,sda(sda0val)(r2)
12                              .end
```

After linking the object file **sdaex.o**, the map file generated by the linker shows the revised section layout of the previous assembly file.

```
Command line: lnkppc -m -o sdaex.x sdaex.o
LOAD sdaex.o

OUTPUT MODULE NAME:    sdaex.x
OUTPUT MODULE FORMAT:  ABSOLUTE
---------------------

SECTION SUMMARY
---------------
SECTION  TYPE  PROGSEG  START    END      SIZE     ALIGN    MODULE
.sdata   DATA  @DATA    00000000 00000003 00000004 8 BYTES  sdaex.o
.sdata2  LIT   @TEXT    00000004 00000007 00000004 8 BYTES  sdaex.o
asmsect  TEXT  @TEXT    00000008 00000017 00000010 8 BYTES  sdaex.o
.PPC.EMB.sdata0
      DATA  @DATA    FFFF8000 FFFF8003 00000003 8 BYTES  sdaex.o

GLOBAL SYMBOL TABLE
------------------
SYMBOL         SECTION     VALUE
```

```
_SDA2_BASE_     (ABSOLUTE)  00000006
_SDA_BASE_      (ABSOLUTE)  00000002
```

For purposes of illustration, the object code generated is shown in terms of an assembly listing. Only the part of the section that is changed is shown in the subsequent listing. Normally, the object code is contained in the resultant absolute file produced by the linker. For simplicity, no other module sections of the same name have been combined.

The resultant object code found in the section **asmsect** of the absolute file would have code equivalent to the following:

```
7                                .sect asmsect
8      00000008 80 AD FF FE    lwz r5, -2 (r13)
9      0000000C 38 ED FF FE    addi r7, r13, -2
10     00000010 81 22 FF FE    lwz r9, -2 (r2)
11     00000014 81 40 80 00    lwz r10, -32768 (r0)
```

Examining line 8 in the previous example, the linker finds that the symbol **sdaval** falls into the small data area **.sdata**. Thus, the linker calculates the signed 16-bit offset of this symbol from:

```
_SDA_BASE_ = 2
sdabase(sdaval) =
sdaval - _SDA_BASE_ = 0 - 2 =  -2 (or 0xFFFE)
```

This value is placed in the 16-bit displacement field of the **lwz** instruction. Also, the linker sets the next 5 bits of this instruction—the base register field of the **lwz** instruction—to 0b01101 (binary). This is **gpr13**, the base register of the **.sdata** section. The same fields are changed similarly for the **addi** instruction in line 9.

Line 10 shows how the linker finds the symbol **sda2val** to be relative to the **.sdata2** section and thus calculates the offset of this symbol from:

```
_SDA2_BASE_ = 6
sda2base(sda2val)=
sda2val - _SDA2_BASE_ = 0x4 - 0x6 = -2 (or 0xFFFE)
```

This value is placed into the 16-bit displacement field of the **lwz** instruction. In addition, the linker sets the next 5 bits of this instruction — the base register field of the **lwz** instruction — to 0b00010 (binary). This is **gpr2**, the base register of the **.sdata2** section.

Finally, line 11 shows how the linker finds the symbol **sda0val** to be relative to the **.PPC.EMB.sdata0** section and calculates the offset of this symbol from zero:

```
sda0val - 0 = 0xFFFF8000 - 0 = 0xFFFF8000 (base 16)
         = -32768 - 0 = -32768 (base 10)
```

This last operation modifies the 16-bit displacement field of the **lwz** instruction. The linker applies **lo(0xFFFF8000) = 0x8000** and stores this result into the 16-bit field. In addition, the linker sets the next 5 bits of this instruction—the base register field of the **lwz** instruction—to 0b00000 (binary). This is **gpr0**, which represents address 0.

The second preceding partial listing shows the new register values and the equivalent operators that the linker has performed automatically.

For this code to work appropriately, the **gpr13** and **gpr2** base registers used in this example would need to be initialized to the base values of the appropriate small data areas.

SDAADDR — Offset to Implicit Pointer in .sdata

SDA2ADDR — Offset to Implicit Pointer in .sdata2

Syntax

sdaaddr (*OPR*)

sda2addr (*OPR*)

Description

- **OPR**

  A relocatable operand

With these operators, you can reference data via pointers stored in the small data areas. The use of these operators creates a 4-byte pointer that is the physical address of *OPR*. These pointers are created by the linker, and their creation is transparent to the user. By accessing these linker-generated pointers via these operators, you can load a full 32-bit address through a single load instruction.

The **sdaaddr()** operator yields a signed 16-bit offset from **_SDA_BASE_** to the location in the **.sdata** section where the linker placed the address of the symbol used in *OPR*.

The **sda2addr()** operator yields a signed 16-bit offset from **_SDA2_BASE_** to the location in the **.sdata2** section where the linker placed the address of the symbol used in *OPR*.

As discussed previously, the linker creates a 4-byte word-aligned entry in the **.sdata** or **.sdata2** section, according to the operator used. In this entry, the linker stores the final address of the relocatable symbol used in *OPR*. The linker creates only one such entry per operand expression in the final executable file. Many modules may use the given operator with the exact same relocatable symbol reference, but only one pointer is created in the respective section.

OPR must be a relocatable symbol.

If *OPR* is valid, the assembler emits special relocatable information to the object file. During an absolute link, the linker will read this information and create the entries and sections specified (that is, only if the *REL* used in the expression has been properly defined).

Notes

If either of the small data areas has not been allocated at link time, the linker will automatically create these sections. Otherwise, the linker will create address entries in the existing small data areas, increasing the size of these small data sections according to the number of unique entries added.

Example

The following example illustrates how the assembler and linker work together to create the pointers in the small data areas. One assembler module uses both operators and the small data areas are defined in the following listing:

```
Command line: asmppc -l asdaptr.s
Line   Address  Opcode           Source
1                                .extern s1, s2
2                                .sect sdarefs
3      00000000 80 CD 00 00      E lwz r6,sdaaddr(s1)(rsda)
4      00000004 80 C2 00 00      E lwz r6,sda2addr(s2)(rsda2)
5
6                                .sect data
7      00000000                  .space 0x10000
8      00010000 FF FF FF F1      s1: .long 0xfffffff1
9      00010004 FF FF FF F2      s2: .long 0xfffffff2
10
11                               .sect .sdata
12     00000000 F3               .byte 0xf3
13
14                               .sect .sdata2
15     00000000 FF FF FF F4      .long 0xfffffff4
16     00000004 F5               .byte 0xf5
```

For purposes of illustration, the object code generated is shown in terms of an assembly listing. Normally, the object code is contained in the resultant absolute file produced by the linker. In the following example, no other module sections of the same name have been combined together.

```
Example
Line   Address  Opcode           Source
1                                .extern s1, s2
2                                .sect sdarefs
3     00000014    80 CD 00 02      lwz r6,sdabase(s1ent)(rsda)
4     00000018    80 C2 00 04    lwz r6,sda2base(s2ent)(rsda2)
5
6                                .sect data
7     0000001C                   .space 0x10000
8     0001001C    FF FF FF F1    s1: .long 0xfffffff1
9     00010020    FF FF FF F2    s2: .long 0xfffffff2
10
11                               .sect .sdata
12    00000000    F3 00          .byte 0xf3, 0x0
13    00000002                   _SDA_BASE_:
14    00000002    00 00          .align 4
15    00000004    00 01 00 1C    s1ent: .long 0x1001c
16
17                               .sect .sdata2
18    00000008    FF FF FF F4    .long 0xfffffff4
19    0000000C                   _SDA2_BASE_:
20    0000000C    F5             .byte 0xf5
21    0000000D                   .align 4
22    0000000E    00 00 00
23    00000010    00 01 00 20    s2ent: .long 0x10020
```

In the above listing, note that the linker created two entries at the end of the Small Data Areas (SDA). At the end of **.sdata** is **s1ent**. A word-aligned 4-byte entry was created to hold the address of symbol **s1**. A similar entry for symbol **s2** was created at the end of **.sdata2**. On line 3, the linker calculated the offset of the **s1ent** from the symbol **_SDA_BASE_**. On line 4, the linker calculated the offset of **s2ent** from the symbol **_SDA2_BASE_**. Those offsets were stored into the 16-bit fields of the appropriate LWZ instruction.

SDAOFF — Calculates 16-bit Offset From an Implicit Small Data Area Base

Syntax

sdaoff (*OPR*)

Description

- **OPR**

  A relocatable integer operand

The **sdaoff()** operator, like the **sda()** operator, gives the signed 16-bit offset of the symbol in *OPR* from:

- **_SDA_BASE_**, if the symbol is in **.sdata** or **.sbss**
- **_SDA2_BASE_**, if the symbol is in **.sdata2** or **.sbss2**
- Zero (0), if the symbol is in **.PPC.EMB.sdata0** or **.PPC.EMB.sbss0**

If the symbol is not found in one of these sections, the link will fail.

This operator does not affect the RA field of the instruction, unlike the **sda()** operator. In addition, the restrictions placed on usage of the **sda()** operator do not hold for **sdaoff()**. This operator can be used in any data directive or instruction that can receive a 16-bit value.

OPR must be an expression that conforms to the following context free grammar:

```
OPR          ::= VAL2

VAL2         ::= REL | REL + ABS | REL - ABS | ABS + REL
```

ABS denotes an absolute symbol, constant, or expression. *REL* denotes a relocatable symbol or expression. If the expression does not conform to the above form, an error message is generated by the assembler.

If *OPR* is valid, the assembler emits special relocatable information to the object file. During an absolute link, the linker will read this information and calculate the final address values (only if the *REL* used in the expression has been properly defined). The usage of this operator affects the 16-bit field where it is used. The linker performs a range check to validate whether the offset fits within the 16-bit field. If not, a link-time error is produced.

Notes

When used with the **.long** data directive, the result is placed into either the lower 16 bits or upper 16 bits of the 32-bit field, depending upon the byte order setting. See Chapter 6, “Assembler Directives”, and the **.endian** directive for more information on byte ordering.

Example

The following assembly listing is derived from the example used for the **sda()** operator. The base registers used in the instruction do not change.

```
Command line: asmppc -l sdaoff.s
Line   Address  Opcode         Source
1                              .sect .sdata
2      00000000 00 00 00 01    sdaval: .long 1
3                              .sect .sdata2
4      00000000 00 00 00 02    sda2val: .long 2
5                              .sect .PPC.EMB.sdata0
6      00000000 00 00 00 03    sda0val: .long 3
7                              .sect asmsect
8      00000000 80 B2 00 00  R lwz r5, sdaoff(sdaval)(r18)
9      00000004 38 E8 00 00  R addi r7, r8, sdaoff(sdaval)
10     00000008 81 33 00 00  R lwz r9, sdaoff(sda2val)(r19)
11     0000000C 81 42 00 00  R lwz r10, sdaoff(sda0val)(r2)
```

If the final link produces the same values as shown in the **sda()** operator example, the object code generated for the section **asmsect** would be:

```
8      00000008 80 B2 FF FE  R lwz r5, -2 (r18)
9      0000000C 38 E8 FF FE  R addi r7, r8, -2
10     00000010 81 33 FF FE  R lwz r9, -2 (r19)
11     00000014 81 42 80 00  R lwz r10, -32768 (r2)
```

The only difference is that the **sdaoff()** operator will not cause the base register fields to change. This operator only affects the 16-bit displacement field of the instruction.

In order for this code to work correctly, the source/destination registers used in this example would need to be initialized to the base values of the appropriate small data areas.

SIZEOF — Specifies Section Size
SIZELO — Low 16 Bits of Section Size
SIZEHI — High 16 Bits of Section Size
SIZEHA — High Adjusted 16 Bits of Section Size

Syntax

sizeof (*section_name*)

sizelo (*section_name*)

sizehi (*section_name*)

sizeha (*section_name*)

Description

- **section_name**

  Specifies a section name

The **sizeof()** operator returns the size of *section_name*. The result is a 32-bit value.

The **sizelo()** operator yields the low 16 bits of the size of *section_name*.

The **sizehi()** operator gives the high 16 bits of the size of *section_name*.

The **sizeha()** operator gives the high adjusted 16 bits of the size of *section_name*.

All of these operators give a relocatable value, which is resolved at link time.

Notes

See the related operators **sectof()**, **sectlo()**, **secthi()**, **sectha()**, **startof()**, **startlo()**, **starthi()**, and **startha()**.

The values returned by these operators may be incorrect if the indicated section is not allocated in a single linear region of memory at absolute link time.

Example

```
#   This routine will clear memory for the full
#   address range of the final combined section
#   that contains the subsection zerovars from
#   this module.

          .sect zerovars
            .byte 1,2,3,4,5,6,7,8,9,0xa,0xb,0xc,0xd,0xe,0xf
          .sect text
          # reg3 = size
          addis   3,0,sizehi(zerovars)
          ori     3,3,sizelo(zerovars)
```

```
        # Load start address of zerovars in reg4
        LD_ADDR 4,startof(zerovars)
        # r3 = size + start
        addc    3,3,4
        # compare size to 0
        cmpl    0,0x0,4,3
        # branch to end if size <= 0
        bc      0x5,0x0,end_init
        # r5 = 0 (init value)
        addc    5,0,0
varsloop:
        stb     5,0x0(4)
        addic   4,4,0x1
        cmpl    0,0x0,3,4
        bc      0xc,0x1,varsloop
end_init:
```

STARTOF — Specifies Section Start Address

STARTLO — Low 16 Bits of Section Start Address

STARTHI — High 16 Bits of Section Start Address

STARTHA — High Adjusted 16 Bits of Section Start Address

Syntax

startof (*section_name*)

startlo (*section_name*)

starthi (*section_name*)

startha (*section_name*)

Description

- **section_name**

  Specifies a section name, which may or may not be defined in the module being assembled.

The **startof()** operator gives the start address of *section_name*. The result is a 32-bit value or a 16-bit value depending on context. For example, with the **.short** directive, a 16-bit relocation is produced; if the section start does not fit in 16 bits, the linker reports an error.

The **startlo()** operator yields the low 16 bits of the start address of *section_name*.

The **starthi()** operator gives the high 16 bits of the start address of *section_name*.

The **startha()** operator gives the high adjusted 16 bits of the start address of *section_name*.

All of these operators give a relocatable value, which is resolved at link time.

If *section_name* is valid or undefined in the module, the assembler emits relocation information to the object file when these operators are used. During an absolute link, the linker will read this information and perform the final operations.

The start address of the section is the memory location of the first byte of the combined section that is created at absolute link time. It is not necessarily the same as the address of the first byte of the section in this module.

Notes

See the related operators **sectof()**, **sectlo()**, **secthi()**, **sectha()**, and **sizeof()**, **sizelo()**, **sizehi()**, **sizeha()**.

See the example for the **sizeof()** and **startof()** operators in the previous section.

Chapter 4
Assembly Instructions

PowerPC Instructions

This chapter lists many of the common PowerPC assembly instructions and pseudo-instructions. A pseudo-instruction is an instruction that represents an alternative method of representing an actual PowerPC assembly instruction. The instructions that are accepted by the Microtec PowerPC toolkit are listed in the following tables:

- Compatible Instructions and Pseudo-Instruction Mnemonics
- Floating-point Instructions

All PowerPC variants will execute instructions or pseudo-instructions listed in Table 4-3. The instructions in Table 4-4 are accepted by those PowerPC variants that have an FPU or if the variant name is modified with the **/F** modifier when specified with the **-p** command line option or with the **.cputype** directive. A PowerPC variant may also accept instructions that are not noted in either of these two instruction tables. A complete list of accepted instructions can be found in the manufacturer's user manual. A list can also be generated using the **v** flag with the **-f** command line option or with the **.lflags** directive.

Instructions are organized within each table in alphabetical order.

Each table of instructions has three columns. The first column lists the instruction or pseudo-instruction mnemonic as documented in the manufacturer's PowerPC user's manuals. The second column shows the operands that are valid for use with that mnemonic. An error will be reported if an instruction's encoding does not follow this guideline. (An explanation of the operand representation is in Table 4-1. Operands must be separated by commas or parentheses, as shown for each instruction.) The third column shows any additional information about the mnemonic that may be useful. For example, if the mnemonic is a pseudo-instruction, the related instruction is shown. Other information, as documented in Table 4-2, is shown if applicable.

Some instructions may accept fewer operands than are shown in the instruction tables. For example, some branch instructions may have an implied first operand (the first operand may be omitted). If such an instruction is processed, the first operand is assumed to have a value of 0. The following is an example of such an instruction and its optional encoding:

```
beq 0,l1
beq l1 # BIF assumed to be 0
```

Another category of instructions that may have a missing operand are the following instructions:

```
mnemonic rD,SIMM(rA)
mnemonic rS,SIMM(rA)
```

For these instructions, an alternative encoding is:

```
mnemonic rD,SIMM
mnemonic rS,SIMM
```

The assumed value of the missing operand is determined by the value of **SIMM**. If **SIMM** is an absolute value or expression, the value of **rA** is assumed to be r0, or 0. If SIMM is a relocatable value or an expression using a symbol from a **dsect** section, then the assembler will check to see if a related expression can be found that has been assigned to a base register through the **.using** directive.

To be related, the **.using** expression must belong to the same section as the **SIMM** expression, and the difference between the two expressions must be less than 32K plus or minus. If such an expression can be found, **rA** is set to the matching base register and SIMM is replaced by the absolute value that represents the difference between the two expressions. If no such expression has been assigned through the **.using** directive, then the assembler cannot assign a value to **rA** and will generate an error message. No additional processing of an instruction occurs if the **rA** register is indicated directly.

If two base registers are found that are related to the same expression, the closer base register will be used. If both registers are the same distance, the lower numbered register will be used.

Notational Conventions

The notational conventions explained in this section are used in the instruction tables in sections that follow it.

Operands

Table 4-1 contains a list of all operand forms that are used in describing valid instruction syntax. Any use of a symbol from the first column represents an occurrence of a symbol or value that is described in the second column.

Table 4-1. Notational Conventions - Operands

Operands	**Meaning**
b	Absolute value. Limitations on the value will vary.
BI	Absolute value from 0 to 31 inclusive. Each bit in the value, as represented in binary form, represents the bit in the condition register to be tested.

Table 4-1. Notational Conventions - Operands (cont.)

Operands	**Meaning**
BIF	Condition register. Must be either a condition register symbol (such as **cr3**) or an absolute value from 0 to 7, inclusive. The bit within the condition register that is to be tested is set depending on the pseudo-instruction.
BO	Absolute value from 0 to 31 inclusive. Each bit in the value, as represented in binary form, represents a condition to be checked for. Various tests include: always branch, branch if true, decrement condition register, and branch on 0.
crbD, crbB, crbA	Absolute value from 0 to 31, inclusive. The value represents one of the bits within the condition register.
CRB	Absolute value from 0 to 31, inclusive. The value represents one of the bits within the condition register.
crbit	Absolute value from 0 to 31, inclusive. The value represents one of the bits within the condition register.
CRFS	Absolute value from 0 to 7, inclusive. The value represents one of the bits within the condition register.
crfD, crfS	Condition register. Must be either a condition register symbol (such as **cr3**) or an absolute value from 0 to 7 inclusive.
CRM	Absolute value from 0 to 255, inclusive. The value represents a mask for bits within the condition register.
CT	Absolute value from 0 to 31, inclusive. Each bit in the value, as represented in binary form, represents a condition that can be tested for.
EXP	Bit mask value. The assembler creates the equivalent MB and ME values by detecting the start and end bit positions of the “1's” in the binary form of the expression.
fD,fA,fB,fS,fC	Floating-point registers. Must be either a floating-point register symbol (such as **f15**) or an absolute value from 0 to 31, inclusive.
FM	Absolute value from 0 to 255, inclusive. The value represents a mask for bits within the floating-point condition register.
L	Flag for 32-/64-bit operation. The only valid value is 0.
MB	Absolute value from 0 to 31, inclusive. Bit offset of start of mask.
ME	Absolute value from 0 to 31, inclusive. Bit offset of end of mask.
MO	Absolute value from 0 to 31, inclusive. Each bit in the value, as represented in binary form, represents a condition that can be tested for.
n	Absolute value. Limitations on the value will vary.

Table 4-1. Notational Conventions - Operands (cont.)

Operands	Meaning
NB	Absolute value from 0 to 31 inclusive. This value represents the number of bytes that are to be moved.
NSIMM	16-bit signed value, whose value will be negated.
NSIMM128	17-bit signed value, whose value will be negated.
rD, rA, rB, rS	General registers. Must be either a general register symbol (such as **r15**) or an absolute value from 0 to 31 inclusive.
REL16	16-bit value, as represented by either an absolute value or a label. The resulting value must be divisible by 4.
REL16_PC	16-bit value, as represented by a label. The resulting value will be calculated as the difference between the location of this instruction and the destination address. The resulting value must be divisible by 4.
REL26	26-bit value, as represented by either an absolute value or a label. The resulting value must be divisible by 4.
REL_26PC	26-bit PC-relative value, as represented by a label. The resulting value will be calculated as the difference between the location of this instruction and the destination address. The resulting value must be divisible by 4.
SH	Absolute value from 0 to 31, inclusive. The value represents the number of positions that the result is to be shifted by.
SIMM	16-bit signed value. Also, expressions resulting from the LO/HI/HA operators or integer constants that fit in 16-bits are allowed.
SIM5	5-bit signed value.
SIMM128	17-bit signed value.
SPR	Special purpose register. Must be either a special purpose register symbol (such as **IBAT0L**) or an absolute value from 0 to 1023, inclusive.
TO	Absolute value from 0 to 31 inclusive. Each bit in the value, as represented in binary form, represents a different condition that can be tested for.
UIMM	16-bit unsigned value.
UIMM4	4-bit unsigned value.
UIMM5	5-bit unsigned value
<None>	No operand may be used with this instruction.

Notes

Table 4-2 lists symbols that may appear in the Notes column in the table that describes PowerPC instructions. These notes are used to indicate special behavior that may occur when a particular instruction or pseudo-instruction is used.

Table 4-2. Notational Conventions - Notes

Notes	Meaning
AP	Branch prediction (+ or -) allowed. If used, the plus or minus character must immediately follow the instruction or pseudo-instruction mnemonic.
LE	Execution of this instruction in little-endian mode causes an exception.
RA_RD_EQ_O	Instructions with the form rA = rD = 0 are invalid. If rA is in the range of registers being affected, then the instruction is invalid.
RA_EQ_O	Instructions with rA = 0 are invalid.
RA_EQ_RD	Instructions with rA = rD are invalid.
SUPERVISOR	Instruction valid only in supervisor mode. If seen, produces a warning in the assembler if the assembler assumes the code will be executed in user mode.
SIM_REM	Instruction may be made invalid if the **/NF** modifier is used when declaring the processor type.
TP	First operand is optional. If not used, the value of the first operand is assumed to be 0.

Compatible Instructions

Table 4-3 lists all instructions and pseudo-instruction mnemonics that are common to all of the PowerPC processors. If the instruction is a pseudo-instruction for an actual mnemonic, the equivalent mnemonic is shown in the Notes text. Any instructions listed in this table will execute on any PowerPC processor (with the exception of invalid SPR registers).

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics

(Pseudo)-Instruction	Operands	Notes
ADD	rD,rA,rB	
ADD**.**	rD,rA,rB	
ADDC	rD,rA,rB	
ADDC**.**	rD,rA,rB	
ADDCO	rD,rA,rB	

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics (cont.)

(Pseudo)-Instruction	Operands	Notes
ADDCO.	rD,rA,rB	
ADDE	rD,rA,rB	
ADDE.	rD,rA,rB	
ADDEO	rD,rA,rB	
ADDEO.	rD,rA,rB	
ADDI	rD,rA,SIMM	
ADDIC	rD,rA,SIMM	
ADDIC.	rD,rA,SIMM	
ADDIS	rD,rA,SIMM128	
ADDME	rD,rA	
ADDME.	rD,rA	
ADDMEO	rD,rA	
ADDMEO.	rD,rA	
ADDO	rD,rA,rB	
ADDO.	rD,rA,rB	
ADDZE	rD,rA	
ADDZE.	rD,rA	
ADDZEO	rD,rA	
ADDZEO.	rD,rA	
AND	rA,rS,rB	
AND.	rA,rS,rB	
ANDC	rA,rS,rB	
ANDC.	rA,rS,rB	
ANDI.	rA,rS,UIMM	
ANDIS.	rA,rS,UIMM	
B	REL26_PC	
BA	REL26	
BC	BO,BI,REL16_PC	AP
BCA	BO,BI,REL16	AP
BCCTR	BO,BI	AP

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics (cont.)

(Pseudo)-Instruction	Operands	Notes
BCCTRL	BO,BI	AP
BCL	BO,BI,REL16_PC	AP
BCLA	BO,BI,REL16	AP
BCLR	BO,BI	AP
BCLRL	BO,BI	AP
BCTR	<NONE>	BCCTR 20,0 AP
BCTRL	<NONE>	BCCTRL 20,0 AP
BDNZ	BI,REL16_PC	BC 16,BI,REL16_PC AP
BDNZA	BI,REL16	BCA 16,BI,REL16 AP
BDNZF	BI,REL16_PC	BC 0,BI,REL16_PC AP | TP
BDNZFA	BI,REL16	BCA 0,BI,REL16 AP | TP
BDNZFL	BI,REL16_PC	BCL 0,BI,REL16_PC AP | TP
BDNZFLA	BI,REL16	BCLA 0,BI,REL16 AP | TP
BDNZFLR	BI	BCLR 0,BI AP | TP
BDNZFLRL	BI	BCLRL 0,BI AP | TP
BDNZL	BI,REL16_PC	BCL 16,BI,REL16_PC AP
BDNZLA	BI,REL16	BCLA 16,BI,REL16 AP
BDNZLR	<NONE>	BCLR 16,0 AP
BDNZLRL	<NONE>	BCLRL 16,0 AP
BDNZT	BI,REL16_PC	BC 8,BI,REL16_PC AP | TP

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics (cont.)

(Pseudo)-Instruction	Operands	Notes
BDNZTA	BI,REL16	BCA 8,BI,REL16 AP | TP
BDNZTL	BI,REL16_PC	BCL 8,BI,REL16_PC AP | TP
BDNZTLA	BI,REL16	BCLA 8,BI,REL16 AP | TP
BDNZTLR	BI	BCLR 8,BI AP | TP
BDNZTLRL	BI	BCLRL 8,BI AP | TP
BDZ	BI,REL16_PC	BC 18,BI,REL16_PC AP
BDZA	BI,REL16	BC 18,BI,REL16 AP
BDZF	BI,REL16_PC	BC 2,BI,REL16_PC AP | TP
BDZFA	BI,REL16	BCA 2,BI,REL16 AP | TP
BDZFL	BI,REL16_PC	BCL 2,BI,REL16_PC AP | TP
BDZFLA	BI,REL16	BCLA 2,BI,REL16 AP | TP
BDZFLR	BI	BCLR 2,BI AP | TP
BDZFLRL	BI	BCLRL 2,BI AP | TP
BDZL	BI,REL16_PC	BCL 18,BI,REL16_PC AP
BDZLA	BI,REL16	BCLA 18,BI,REL16 AP
BDZLR	<NONE>	BCLR 18,0 AP
BDZLRL	<NONE>	BCLRL 18,0 AP
BDZT	BI,REL16_PC	BC 10,BI,REL16_PC AP | TP
BDZTA	BI,REL16	BCA 10,BI,REL16 AP | TP

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics (cont.)

(Pseudo)-Instruction	Operands	Notes
BDZTL	BI,REL16_PC	BCL 10,BI,REL16_PC AP | TP
BDZTLA	BI,REL16	BCLA 10,BI,REL16 AP | TP
BDZTLR	BI	BCLR 10,BI AP | TP
BDZTLRL	BI	BCLRL 10,BI AP | TP
BEQ	BIF,REL16_PC	BC 12,2+4*BIF,REL16_PC AP | TP
BEQA	BIF,REL16	BCA 12,2+4*BIF,REL16 AP | TP
BEQCTR	BIF	BCCTR 12,2+4*BIF AP | TP
BEQCTRL	BIF	BCCTRL 12,2+4*BIF AP | TP
BEQL	BIF,REL16_PC	BCL 12,2+4*BIF,REL16_PC AP | TP
BEQLA	BIF,REL16	BCLA 12,2+4*BIF,REL16 AP | TP
BEQLR	BIF	BCLR 12,2+4*BIF AP | TP
BEQLRL	BIF	BCLRL 12,2+4*BIF AP | TP
BF	BI,REL16_PC	BC 4,BI,REL16_PC AP | TP
BFA	BI,REL16	BCA 4,BI,REL16 AP | TP
BFCTR	BI	BCCTR 4,BI AP | TP
BFCTRL	BI	BCCTRL 4,BI AP | TP
BFL	BI,REL16_PC	BCL 4,BI,REL16_PC AP | TP
BFLA	BI,REL16	BCLA 4,BI,REL16 AP | TP

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics (cont.)

(Pseudo)-Instruction	Operands	Notes
BFLR	BI	BCLR 4,BI AP | TP
BFLRL	BI	BCLRL 4,BI AP | TP
BGE	BIF,REL16_PC	BC 4,4*BIF,REL16_PC AP | TP
BGEA	BIF,REL16	BCA 4,4*BIF,REL16 AP | TP
BGECTR	BIF	BCCTR 4,4*BIF AP | TP
BGECTRL	BIF	BCCTRL 4,4*BIF AP | TP
BGEL	BIF,REL16_PC	BCL 4,4*BIF,REL16_PC AP | TP
BGELA	BIF,REL16	BCLA 4,4*BIF,REL16 AP | TP
BGELR	BIF	BCLR 4,4*BIF AP | TP
BGELRL	BIF	BCLRL 4,4*BIF AP | TP
BGT	BIF,REL16_PC	BC 12,1+4*BIF,REL16_PC AP | TP
BGTA	BIF,REL16	BCA 12,1+4*BIF,REL16 AP | TP
BGTCTR	BIF	BCCTR 12,1+4*BIF AP | TP
BGTCTRL	BIF	BCCTRL 12,1+4*BIF AP | TP
BGTL	BIF,REL16_PC	BCL 12,1+4*BIF,REL16_PC AP | TP
BGTLA	BIF,REL16	BCLA 12,1+4*BIF,REL16 AP | TP
BGTLR	BIF	BCLR 12,1+4*BIF AP | TP
BGTLRL	BIF	BCLRL 12,1+4*BIF AP | TP
BL	REL26_PC	

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics (cont.)

(Pseudo)-Instruction	Operands	Notes
BLA	REL26	
BLE	BIF,REL16_PC	BC 4,1+4*BIF,REL16_PC AP | TP
BLEA	BIF,REL16	BCA 4,1+4*BIF,REL16 AP | TP
BLECTR	BIF	BCCTR 4,1+4*BIF AP | TP
BLECTRL	BIF	BCCTRL 4,1+4*BIF AP | TP
BLEL	BIF,REL16_PC	BCL 4,1+4*BIF,REL16_PC AP | TP
BLELA	BIF,REL16	BCLA 4,1+4*BIF,REL16 AP | TP
BLELR	BIF	BCLR 4,1+4*BIF AP | TP
BLELRL	BIF	BCLRL 4,1+4*BIF AP | TP
BLR	<NONE>	BCLR 20,0 AP
BLRL	<NONE>	BCLRL 20,0 AP
BLT	BIF,REL16_PC	BC 12,4*BIF,REL16_PC AP | TP
BLTA	BIF,REL16	BCA 12,4*BIF,REL16 AP | TP
BLTCTR	BIF	BCCTR 12,4*BIF AP | TP
BLTCTRL	BIF	BCCTRL 12,4*BIF AP | TP
BLTL	BIF,REL16_PC	BCL 12,4*BIF,REL16_PC AP | TP
BLTLA	BIF,REL16	BCLA 12,4*BIF,REL16 AP | TP
BLTLR	BIF	BCLR 12,4*BIF AP | TP
BLTLRL	BIF	BCLRL 12,4*BIF AP | TP

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics (cont.)

(Pseudo)-Instruction	Operands	Notes
BNE	BIF,REL16_PC	BC 4,2+4*BIF,REL16_PC AP | TP
BNEA	BIF,REL16	BCA 4,2+4*BIF,REL16 AP | TP
BNECTR	BIF	BCCTR 4,2+4*BIF AP | TP
BNECTRL	BIF	BCCTRL 4,2+4*BIF AP | TP
BNEL	BIF,REL16_PC	BCL 4,2+4*BIF,REL16_PC AP | TP
BNELA	BIF,REL16	BCLA 4,2+4*BIF,REL16 AP | TP
BNELR	BIF	BCLR 4,2+4*BIF AP | TP
BNELRL	BIF	BCLRL 4,2+4*BIF AP | TP
BNG	BIF,REL16_PC	BC 4,1+4*BIF,REL16_PC AP | TP
BNGA	BIF,REL16	BCA 4,1+4*BIF,REL16 AP | TP
BNGCTR	BIF	BCCTR 4,1+4*BIF AP | TP
BNGCTRL	BIF	BCCTRL 4,1+4*BIF AP | TP
BNGL	BIF,REL16_PC	BCL 4,1+4*BIF,REL16_PC AP | TP
BNGLA	BIF,REL16	BCLA 4,1+4*BIF,REL16 AP | TP
BNGLR	BIF	BCLR 4,1+4*BIF AP | TP
BNGLRL	BIF	BCLRL 4,1+4*BIF AP | TP
BNL	BIF,REL16_PC	BC 4,4*BIF,REL16_PC AP | TP
BNLA	BIF,REL16	BCA 4,4*BIF,REL16 AP | TP

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics (cont.)

(Pseudo)-Instruction	Operands	Notes
BNLCTR	BIF	BCCTR 4,4*BIF AP | TP
BNLCTRL	BIF	BCCTRL 4,4*BIF AP | TP
BNLL	BIF,REL16_PC	BCL 4,4*BIF,REL16_PC AP | TP
BNLLA	BIF,REL16	BCLA 4,4*BIF,REL16 AP | TP
BNLLR	BIF	BCLR 4,4*BIF AP | TP
BNLLRL	BIF	BCLRL 4,4*BIF AP | TP
BNS	BIF,REL16_PC	BC 4,3+4*BIF,REL16_PC AP | TP
BNSA	BIF,REL16	BCA 4,3+4*BIF,REL16 AP | TP
BNSCTR	BIF	BCCTR 4,3+4*BIF P | TP
BNSCTRL	BIF	BCCTRL 4,3+4*BIF AP | TP
BNSL	BIF,REL16_PC	BCL 4,3+4*BIF,REL16_PC AP | TP
BNSLA	BIF,REL16	BCLA 4,3+4*BIF,REL16 AP | TP
BNSLR	BIF	BCLR 4,3+4*BIF AP | TP
BNSLRL	BIF	BCLRL 4,3+4*BIF AP | TP
BNU	BIF,REL16_PC	BC 4,3+4*BIF,REL16_PC AP | TP
BNUA	BIF,REL16	BCA 4,3+4*BIF,REL16 AP | TP
BNUCTR	BIF	BCCTR 4,3+4*BIF AP | TP
BNUCTRL	BIF	BCCTRL 4,3+4*BIF AP | TP

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics (cont.)

(Pseudo)-Instruction	Operands	Notes
BNUL	BIF,REL16_PC	BCL 4,3+4*BIF,REL16_PC AP | TP
BNULA	BIF,REL16	BCLA 4,3+4*BIF,REL16 AP | TP
BNULR	BIF	BCLR 4,3+4*BIF AP | TP
BNULRL	BIF	BCLRL 4,3+4*BIF AP | TP
BSO	BIF,REL16_PC	BC 12,3+4*BIF,REL16_PC AP | TP
BSOA	BIF,REL16	BCA 12,3+4*BIF,REL16 AP | TP
BSOCTR	BIF	BCCTR 12,3+4*BIF AP | TP
BSOCTRL	BIF	BCCTRL 12,3+4*BIF AP | TP
BSOL	BIF,REL16_PC	BCL 12,3+4*BIF,REL16_PC AP | TP
BSOLA	BIF,REL16	BCLA 12,3+4*BIF,REL16 AP | TP
BSOLR	BIF	BCLR 12,3+4*BIF AP | TP
BSOLRL	BIF	BCLRL 12,3+4*BIF AP | TP
BT	BI,REL16_PC	BC 12,BI,REL16_PC AP | TP
BTA	BI,REL16	BCA 12,BI,REL16 AP | TP
BTCTR	BI	BCCTR 12,BI AP | TP
BTCTRL	BI	BCCTRL 12,BI AP | TP
BTL	BI,REL16_PC	BCL 12,BI,REL16_PC AP | TP
BTLA	BI,REL16	BCLA 12,BI,REL16 AP | TP

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics (cont.)

(Pseudo)-Instruction	Operands	Notes
BTLR	BI	BCLR 12,BI AP | TP
BTLRL	BI	BCLRL 12,BI AP | TP
BUN	BIF,REL16_PC	BC 12,3+4*BIF,REL16_PC AP | TP
BUNA	BIF,REL16	BCA 12,3+4*BIF,REL16 AP | TP
BUNCTR	BIF	BCCTR 12,3+4*BIF AP | TP
BUNCTRL	BIF	BCCTRL 12,3+4*BIF AP | TP
BUNL	BIF,REL16_PC	BCL 12,3+4*BIF,REL16_PC AP | TP
BUNLA	BIF,REL16	BCLA 12,3+4*BIF,REL16 AP | TP
BUNLR	BIF	BCLR 12,3+4*BIF AP | TP
BUNLRL	BIF	BCLRL 12,3+4*BIF AP | TP
CLRLSLWI	rA,rS,b,n	RLWINM rA,rS,n,b-n,31-n
CLRLSLWI**.**	rA,rS,b,n	RLWINM. rA,rS,n,b-n,31-n
CLRLWI	rA,rS,n	RLWINM rA,rS,0,n,31
CLRLWI**.**	rA,rS,n	RLWINM. rA,rS,0,n,31
CLRRWI	rA,rS,n	RLWINM rA,rS,0,0,31-n
CLRRWI**.**	rA,rS,n	RLWINM. rA,rS,0,0,31-n
CMP	crfD,L,rA,rB	
	L,rA,rB	CMP 0,L,rA,rB
CMPI	crfD,L,rA,SIMM	
	L,rA,SIMM	CMPI 0,L,rA,SIMM
CMPL	crfD,L,rA,rB	
	L,rA,rB	CMPL 0,L,rA,rB
CMPLI	crfD,L,rA,UIMM	

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics (cont.)

(Pseudo)-Instruction	Operands	Notes
	L,rA,UIMM	CMPLI 0,L,rA,UIMM
CMPLW	crfD,rA,rB	CMPL crfd,0,rA,rB
	rA,rB	CMPL 0,0,rA,rB
CMPLWI	crfD,rA,UIMM	CMPLI crfd,0,rA,rB
	rA,UIMM	CMPLI 0,0,rA,rB
CMPW	crfD,rA,rB rA,rB	CMP crfd,0,rA,rB CMP 0,0,rA,rB
CMPWI	crfD,rA,SIMM rA,SIMM	CMPI crfd,0,rA,rB CMPI 0,0,rA,rB
CNTLZW	rA,rS	
CNTLZW**.**	rA,rS	
CRAND	crbD,crbA,crbB	
CRANDC	crbD,crbA,crbB	
CRCLR	crbit	CRXOR crbit,crbit,crbit
CREQV	crbD,crbA,crbB	
CRMOVE	crbD,crbit	CROR crbD,crbit,crbit
CRNAND	crbD,crbA,crbB	
CRNOR	crbD,crbA,crbB	
CRNOT	crbD,crbit	CRNOR crbD,crbit,crbit
CROR	crbD,crbA,crbB	
CRORC	crbD,crbA,crbB	
CRSET	crbit	CREQV crbit,crbit,crbit
CRXOR	crbD,crbA,crbB	
DIVW	rD,rA,rB	
DIVW**.**	rD,rA,rB	
DIVWO	rD,rA,rB	
DIVWO**.**	rD,rA,rB	
DIVWU	rD,rA,rB	
DIVWU**.**	rD,rA,rB	
DIVWUO	rD,rA,rB	

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics (cont.)

(Pseudo)-Instruction	Operands	Notes
DIVWUO.	rD,rA,rB	
EQV	rA,rS,rB	
EQV.	rA,rS,rB	
EXTLWI	rA,rS,n,b	RLWINM rA,rS,b,0,n-1
EXTLWI.	rA,rS,n,b	RLWINM. rA,rS,b,0,n-1
EXTRWI	rA,rS,n,b	RLWINM rA,rS,b+n,32-n,31
EXTRWI.	rA,rS,n,b	RLWINM. rA,rS,b+n,32-n,31
EXTSB	rA,rS	
EXTSB.	rA,rS	
EXTSH	rA,rS	
EXTSH.	rA,rS	
INSLWI	rA,rS,n,b	RLWIMI rA,rS,32-b,b,b+n-1
INSLWI.	rA,rS,n,b	RLWIMI rA,rS,32-b,b,b+n-1
INSRWI	rA,rS,n,b	RLWIMI rA,rS,32-(b+n),b,b+n-1
INSRWI.	rA,rS,n,b	RLWIMI. rA,rS,32(b+n),b,b+n-1
ISYNC	<NONE>	
LA	rD,SIMM(rA)	ADDI rD,rA,SIMM
LBZ	rD,SIMM(rA)	
LBZU	rD,SIMM(rA)	RA_EQ_0 | RA_EQ_RD
LBZUX	rD,rA,rB	RA_EQ_0 | RA_EQ_RD
LBZX	rD,rA,rB	
LHA	rD,SIMM(rA)	
LHAU	rD,SIMM(rA)	RA_EQ_0 | RA_EQ_RD
LHAUX	rD,rA,rB	RA_EQ_0 | RA_EQ_RD
LHAX	rD,rA,rB	
LHBRX	rD,rA,rB	

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics (cont.)

(Pseudo)-Instruction	Operands	Notes
LHZ	rD,SIMM(rA)	
LHZU	rD,SIMM(rA)	RA_EQ_0 | RA_EQ_RD
LHZUX	rD,rA,rB	RA_EQ_0 | RA_EQ_RD
LHZX	rD,rA,rB	
LI	rD,SIMM	ADDI rD,0,SIMM
LIS	rD,SIMM128	ADDIS rD,0,SIMM128
LMW	rD,SIMM(rA)	RA_RD_EQ_0 LE
LWARX	rD,rA,rB	
LWBRX	rD,rA,rB	
LWZ	rD,SIMM(rA)	
LWZU	rD,SIMM(rA)	RA_EQ_0 | RA_EQ_RD
LWZUX	rD,rA,rB	RA_EQ_0 | RA_EQ_RD
LWZX	rD,rA,rB	
MCRF	crfD,crfS	
MCRXR	crfD	
MFCR	rD	
MFCTR	rD	MFSPR rD,9
MFLR	rD	MFSPR rD,8
MFMSR	rD	SUPERVISOR
MFPVR	rD	MFSPR rD,287 SUPERVISOR
MFSPR	rD,SPR	
MFSPRG	rD,n	MFSPR rD,272+n SUPERVISOR
MFSRR0	rD	MFSPR rD,26 SUPERVISOR
MFSRR1	rD	MFSPR rD,27 SUPERVISOR
MFXER	rD	MFSPR rD,1
MR	rA,rX	OR rA,rX,rX

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics (cont.)

(Pseudo)-Instruction	Operands	Notes
MR.	rA,rX	OR. rA,rX,rX
MTCR	rS	MTCRF 0xff,rS
MTCRF	CRM,rS	
MTCTR	rS	MTSPR 9,rS
MTLR	rS	MTSPR 8,rS
MTMSR	rS	SUPERVISOR
MTSPR	SPR,rS	
MTSPRG	n,rS	MTSPR 272+n,rS SUPERVISOR
MTSRR0	rS	MTSPR 26,rS SUPERVISOR
MTSRR1	rS	MTSPR 27,rS SUPERVISOR
MTXER	rS	MTSPR 1,rS
MULHW	rD,rA,rB	
MULHW.	rD,rA,rB	
MULHWU	rD,rA,rB	
MULHWU.	rD,rA,rB	
MULLI	rD,rA,SIMM	
MULLW	rD,rA,rB	
MULLW.	rD,rA,rB	
MULLWO	rD,rA,rB	
MULLWO.	rD,rA,rB	
NAND	rA,rS,rB	
NAND.	rA,rS,rB	
NEG	rD,rA	
NEG.	rD,rA	
NEGO	rD,rA	
NEGO.	rD,rA	
NOP	<NONE>	ORI 0,0,0
NOR	rA,rS,rB	

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics (cont.)

(Pseudo)-Instruction	Operands	Notes
NOR.	rA,rS,rB	
NOT	rA,rS	NOR rA,rS,rS
NOT.	rA,rS	NOR. rA,rS,rS
OR	rA,rS,rB	
OR.	rA,rS,rB	
ORC	rA,rS,rB	
ORC.	rA,rS,rB	
ORI	rA,rS,UIMM	
ORIS	rA,rS,UIMM	
RFI	<NONE>	SUPERVISOR
RLWIMI	rA,rS,SH,EXP	
	rA,rS,SH,MB,ME	
RLWIMI.	rA,rS,SH,EXP	
	rA,rS,SH,MB,ME	
RLWINM	rA,rS,SH,EXP	
	rA,rS,SH,MB,ME	
RLWINM.	rA,rS,SH,EXP	
	rA,rS,SH,MB,ME	
RLWNM	rA,rS,rB,EXP	
	rA,rS,rB,MB,ME	
RLWNM.	rA,rS,rB,EXP	
	rA,rS,rB,MB,ME	
ROTLW	rA,rS,rB	RLWNM rA,rS,rB,0,31
ROTLW.	rA,rS,rB	RLWNM. rA,rS,rB,0,31
ROTLWI	rA,rS,n	RLWINM rA,rS,n,0,31
ROTLWI.	rA,rS,n	RLWINM. rA,rS,n,0,31
ROTRWI	rA,rS,n	RLWINM rA,rS,32-n,0,31
ROTRWI.	rA,rS,n	RLWINM. rA,rS,32-n,0,31
SC	<NONE>	
SLW	rA,rS,rB	

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics (cont.)

(Pseudo)-Instruction	Operands	Notes
SLW.	rA,rS,rB	
SLWI	rA,rS,n	RLWINM rA,rS,n,0,31-n
SLWI.	rA,rS,n	RLWINM. rA,rS,n,0,31-n
SRAW	rA,rS,rB	
SRAW.	rA,rS,rB	
SRAWI	rA,rS,SH	
SRAWI.	rA,rS,SH	
SRW	rA,rS,rB	
SRW.	rA,rS,rB	
SRWI	rA,rS,n	RLWINM rA,rS,32-n,n,31
SRWI.	rA,rS,n	RLWINM. rA,rS,32-n,n,31
STB	rS,SIMM(rA)	
STBU	rS,SIMM(rA)	RA_EQ_0
STBUX	rS,rA,rB	RA_EQ_0
STBX	rS,rA,rB	
STH	rS,SIMM(rA)	
STHBRX	rS,rA,rB	
STHU	rS,SIMM(rA)	RA_EQ_0
STHUX	rS,rA,rB	RA_EQ_0
STHX	rS,rA,rB	
STMW	rS,SIMM(rA)	LE
STW	rS,SIMM(rA)	
STWBRX	rS,rA,rB	
STWCX.	rS,rA,rB	
STWU	rS,SIMM(rA)	RA_EQ_0
STWUX	rS,rA,rB	RA_EQ_0
STWX	rS,rA,rB	
SUB	rD,rB,rA	SUBF rD,rA,rB
SUB.	rD,rB,rA	SUBF. rD,rA,rB
SUBC	rD,rB,rA	SUBFC rD,rA,rB

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics (cont.)

(Pseudo)-Instruction	Operands	Notes
SUBC.	rD,rB,rA	SUBFC. rD,rA,rB
SUBCO	rD,rB,rA	SUBFCO rD,rA,rB
SUBCO.	rD,rB,rA	SUBFCO. rD,rA,rB
SUBF	rD,rA,rB	
SUBF.	rD,rA,rB	
SUBFC	rD,rA,rB	
SUBFC.	rD,rA,rB	
SUBFCO	rD,rA,rB	
SUBFCO.	rD,rA,rB	
SUBFE	rD,rA,rB	
SUBFE.	rD,rA,rB	
SUBFEO	rD,rA,rB	
SUBFEO.	rD,rA,rB	
SUBFIC	rD,rA,SIMM	
SUBFME	rD,rA	
SUBFME.	rD,rA	
SUBFMEO	rD,rA	
SUBFMEO.	rD,rA	
SUBFO	rD,rA,rB	
SUBFO.	rD,rA,rB	
SUBFZE	rD,rA	
SUBFZE.	rD,rA	
SUBFZEO	rD,rA	
SUBFZEO.	rD,rA	
SUBI	rD,rA,NSIMM	ADDI rD,rA,-SIMM
SUBIC	rD,rA,NSIMM	ADDIC rD,rA,-SIMM
SUBIC.	rD,rA,NSIMM	ADDIC. rD,rA,-SIMM
SUBIS	rD,rA,NSIMM128	ADDIS rD,rA,-SIMM128
SUBO	rD,rB,rA	SUBFO rD,rA,rB
SUBO.	rD,rB,rA	SUBFO. rD,rA,rB

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics (cont.)

(Pseudo)-Instruction	Operands	Notes
TRAP	<NONE>	TW 31,0,0
TW	TO,rA,rB	
TWEQ	rA,rB	TW 4,rA,rB
TWEQI	rA,SIMM	TWI 4,rA,SIMM
TWGE	rA,rB	TW 12,rA,rB
TWGEI	rA,SIMM	TWI 12,rA,SIMM
TWGT	rA,rB	TW 8,rA,rB
TWGTI	rA,SIMM	TWI 8,rA,SIMM
TWI	TO,rA,SIMM	
TWLE	rA,rB	TW 20,rA,rB
TWLEI	rA,SIMM	TWI 20,rA,SIMM
TWLGE	rA,rB	TW 5,rA,rB
TWLGEI	rA,SIMM	TWI 5,rA,SIMM
TWLGT	rA,rB	TW 1,rA,rB
TWLGTI	rA,SIMM	TWI 1,rA,SIMM
TWLLE	rA,rB	TW 6,rA,rB
TWLLEI	rA,SIMM	TWI 6,rA,SIMM
TWLLT	rA,rB	TW 2,rA,rB
TWLLTI	rA,SIMM	TWI 2,rA,SIMM
TWLNG	rA,rB	TW 6,rA,rB
TWLNGI	rA,SIMM	TWI 6,rA,SIMM
TWLNL	rA,rB	TW 5,rA,rB
TWLNLI	rA,SIMM	TWI 5,rA,SIMM
TWLT	rA,rB	TW 16,rA,rB
TWLTI	rA,SIMM	TWI 16,rA,SIMM
TWNE	rA,rB	TW 24,rA,rB
TWNEI	rA,SIMM	TWI 24,rA,SIMM
TWNG	rA,rB	TW 20,rA,rB
TWNGI	rA,SIMM	TWI 20,rA,SIMM
TWNL	rA,rB	TW 12,rA,rB

Table 4-3. Compatible Instructions and Pseudo-Instruction Mnemonics (cont.)

(Pseudo)-Instruction	Operands	Notes
TWNLI	rA,SIMM	TWI 12,rA,SIMM
XOR	rA,rS,rB	
XOR**.**	rA,rS,rB	
XORI	rA,rS,UIMM	
XORIS	rA,rS,UIMM	

Floating-Point Instructions

Code being linked into an application may use floating-point instructions. Different PowerPC processors have different abilities to process these instructions.

The assembler generates floating-point enabled code if the processor handles floating point or if an emulator is available. Microtec C and C++ compilers use the **-f** option to enable and the **-nf** option to disable the use of floating-point instructions in high-level code. If the option is not included, the compiler will generate floating-point enabled code if it is supported by the processor hardware.

Table 4-4 lists all of the floating-point instructions that are accepted by the assembler. Some processors, such as the 603, normally accept all of these instructions. Others, such as the 860 or EC603e, cannot execute all of these instructions in hardware and must have a floating-point emulator included in the application if the instructions are used. None of these instructions are accepted by the **COM** processor type by default.

Table 4-4. Floating-Point and String Instructions

(Pseudo)-Instruction	Operands	Notes
FABS	fD,fB	
FABS.	fD,fB	
FADD	fD,fA,fB	
FADD.	fD,fA,fB	
FADDS	fD,fA,fB	
FADDS.	fD,fA,fB	
FCMPO	crfD,fA,fB	
FCMPU	crfD,fA,fB	
FCTIW	fD,fB	
FCTIW.	fD,fB	

Table 4-4. Floating-Point and String Instructions (cont.)

(Pseudo)-Instruction	Operands	Notes
FCTIWZ	fD,fB	
FCTIWZ.	fD,fB	
FDIV	fD,fA,fB	
FDIV.	fD,fA,fB	
FDIVS	fD,fA,fB	
FDIVS.	fD,fA,fB	
FMADD	fD,fA,fC,fB	
FMADD.	fD,fA,fC,fB	
FMADDS	fD,fA,fC,fB	
FMADDS.	fD,fA,fC,fB	
FMR	fD,fB	
FMR.	fD,fB	
FMSUB	fD,fA,fC,fB	
FMSUB.	fD,fA,fC,fB	
FMSUBS	fD,fA,fC,fB	
FMSUBS.	fD,fA,fC,fB	
FMUL	fD,fA,fC	
FMUL.	fD,fA,fC	
FMULS	fD,fA,fC	
FMULS.	fD,fA,fC	
FNABS	fD,fB	
FNABS.	fD,fB	
FNEG	fD,fB	
FNEG.	fD,fB	
FNMADD	fD,fA,fC,fB	
FNMADD.	fD,fA,fC,fB	
FNMADDS	fD,fA,fC,fB	
FNMADDS.	fD,fA,fC,fB	
FNMSUB	fD,fA,fC,fB	
FNMSUB.	fD,fA,fC,fB	

Table 4-4. Floating-Point and String Instructions (cont.)

(Pseudo)-Instruction	Operands	Notes
FNMSUBS	fD,fA,fC,fB	
FNMSUBS.	fD,fA,fC,fB	
FRES	fD,fB	
FRES.	fD,fB	
FRSP	fD,fB	
FRSP.	fD,fB	
FRSQRTE	fD,fB	
FRSQRTE.	fD,fB	
FSEL	fD,fA,fC,fB	
FSEL.	fD,fA,fC,fB	
FSUB	fD,fA,fB	
FSUB.	fD,fA,fB	
FSUBS	fD,fA,fB	
FSUBS.	fD,fA,fB	
LFD	fD,SIMM(rA)	
LFDU	fD,SIMM(rA)	RA_EQ_0
LFDUX	fD,rA,rB	RA_EQ_0
LFDX	fD,rA,rB	
LFS	fD,SIMM(rA)	
LFSU	fD,SIMM(rA)	RA_EQ_0
LFSUX	fD,rA,rB	RA_EQ_0
LFSX	fD,rA,rB	
MCRFS	crfD,crfS	
MFFS	fD	
MFFS.	fD	
MTFSB0	crbD	
MTFSB0.	crbD	
MTFSB1	crbD	
MTFSB1.	crbD	
MTFSF	FM,fB	

Table 4-4. Floating-Point and String Instructions (cont.)

(Pseudo)-Instruction	Operands	Notes
MTFSF.	FM,fB	
MTFSFI	crfD,UIMM4	
MTFSFI.	crfD,UIMM4	
STFD	fS,SIMM(rA)	
STFDU	fS,SIMM(rA)	RA_EQ_0
STFDUX	fS,rA,rB	RA_EQ_0
STFDX	fS,rA,rB	
STFIWX	fS,rA,rB	
STFS	fS,SIMM(rA)	
STFSU	fS,SIMM(rA)	RA_EQ_0
STFSUX	fS,rA,rB	RA_EQ_0
STFSX	fS,rA,rB	

Processor-Specific Instructions

All PowerPC microprocessors have some instructions that are not universally executable. These instructions are documented in the manufacturer's User Manual for that microprocessor.

The allowed operands for each microprocessor-specific instruction or pseudo-instruction depends on the encoding of that value. Registers will continue to be accessible by either numeric values or symbolically. The acceptable range for numeric fields depends on how large a value can be encoded in a given instruction and how that value is used. For example, a signed value will have a different set of allowable values than an unsigned value.

It should also be noted that some instructions are only allowable in "Supervisor" mode. As such, the user should be aware that certain instructions or registers may only be used at certain times in a program. The assembler will warn if it detects potential problems.

Predefined Macros

The assembler defines several macros that perform common functions used in coding. This functionality covers loading registers with constants or values from memory, as well as saving values from registers to memory. The macros are expanded at assembly time to several PowerPC assembly instructions that perform the expected operation. A complete list of predefined macros and their usage can be found in Chapter 8, "Macros".

Chapter 5
Relocation

Object modules produced by the ASMPPC Assembler are in a relocatable format. This allows you to write modular programs whose final addresses will be adjusted by the LNKPPC Linker. Individual program modules can be changed without reassembling the entire program, and separate object modules can be linked together into a final program.

Relocatable programming provides the following advantages:

- Actual memory addresses are of no concern until final link time.
- Large programs can be easily separated into smaller modules, developed separately, and then linked together.
- If one module contains an error, only that module needs to be modified and reassembled.
- Once developed, a library of routines can be used by many people.
- The linker will adjust addresses to meet program requirements.

An object module produced by the ASMPPC Assembler is composed of sections that contain either program instructions or data. Section attributes are defined with the **.sect** assembler directive.

Program Sections

To take advantage of relocatability, you must understand the concept of program sections and how separate object modules are linked together. A program section is that part of a program that contains its own location counter and is logically distinct from other sections. At link time, the addresses for each section can be specified separately.

The assembler and linker produce relocatable object modules composed of sections containing either program instructions or data. The following conventions are recommended but are not automatically enforced by the tools.

- Instructions are placed in text sections.
- Initialized data, as produced by **.long**, **.ualong**, **.short**, **.uashort**, **.string**, **.byte**, **.float**, **.uafloat**, **.double**, **.uadouble**, **.space**, and **.vbyte** directives, are placed in data sections.
- Uninitialized data, as produced by the **.space** or other directives, are placed in **bss** sections.

Because **bss** data have no initial values, data will not be written to the output file, but the output file will contain information useful to the linker in allocating addresses for uninitialized data. Detailed information on section types and contents is provided in Chapter 7, “Section Directives”, and Chapter 10, “Linker Operation”.

Relocatable Sections

A section is relocatable if it is declared by the **.sect** directive. The linker will provide both the relocatable section’s final address and addresses for all nonabsolute references within the section. Nonabsolute references are indicated on the assembler listing with special relocation indicators. See Chapter 9, “Sample Assembler Session”, for more information.

Section Attributes

The **.sect** assembler directive lets you explicitly specify a section’s attribute (see Chapter 7, “Section Directives”).

The section attribute provides information about section contents. In general, program code sections contain instructions, data sections contain read/write data items, and ROMable data sections contain read-only data items:

```
.sect A[AX] # Specifies an allocated and executable section
```

If you do not explicitly specify the section attribute, the assembler assigns the section attribute with **executable** and **allocated** attributes to the section.

Typically, a section will contain either instructions or data, but not both. This arrangement allows you to place the sections in a RAM/ROM environment. However, the assembler permits sections containing a mixture of code and data.

Linking Object Modules

The object modules produced by the assembler are combined or linked together by a linker. The linker converts all relocatable addresses into absolute addresses and resolves references from one module to another. Linkage between modules is provided by external definitions and external references. External references are symbols referenced in one module but defined in another module. The linker combines the external definitions from one module with the external references from other modules to obtain the final addresses. A module can contain both external references and definitions.

Relocatable and Absolute Symbols

Each symbol in the assembler has an associated symbol type, which marks the symbol as absolute or relocatable. If the symbol is relocatable, the type also indicates the section where the symbol was defined. A symbol whose value is not dependent upon program origin is absolute.

A symbol whose value changes when the program origin is changed is a relocatable symbol. Absolute and relocatable symbols can both appear in a relocatable program section.

A symbol is absolute if it is:

- Defined to be a constant expression
- Defined (using labels) in dummy sections

A symbol is relocatable if it:

- Appears in the label field of an instruction
- Has been declared by the **.extern** or **.globl** directive with no other declaration
- Is a user-defined relocatable section name
- Refers to the program counter (**$**) while assembling a relocatable section

Relocatable Expressions

The relocatability of an expression is determined by the relocation of the symbols that comprise the expression. Expressions containing external symbols are relocatable. All numeric constants are considered absolute. Relocatable expressions can be combined to produce an absolute expression, a relocatable expression, or, in certain instances, a complex relocatable expression. The following list shows those expressions whose result is relocatable. *ABS* denotes an absolute symbol, constant, or expression, and *REL* denotes a relocatable symbol or expression.

Expressions that produce a relocatable result have a form defined by the context-free grammar:

```
VAL   ::= VAL16  |  VAL1
VAL16  ::=ha (VAL1)  |  hi (VAL1)  |  lo (VAL1)  |
   sda (VAL2) | sdaoff (VAL2) |
   sdabase (VAL2) | sda2base (VAL2) |
      sectoff (VAL2) | sectha (VAL2) |
      secthi (VAL2)  | sectlo (VAL2) | startha (VAL2) |
      starthi (VAL2) | startlo (VAL2) | sizeha (VAL2) |
      sizehi (VAL2) | sizelo (VAL2)
VAL1   ::=VAL2  |  VAL3
VAL2   ::=REL  |  REL + ABS  |  REL - ABS  |  ABS + REL
VAL3   ::=ABS - REL
```

The *VAL2* and *VAL3* expression form syntax is restricted with respect to the type of relocatable operand used for *REL*. The following grammar rules indicate legal syntax accepted by the assembler. When the assembler parses expressions that do not conform to the following grammar rules, an error is generated.

When *REL* is an external or common symbol:

external_expr ::= [*abs_expr* {+|-}] *external_sym_expr*
[{+|-} *abs_expr*]

abs_expr ::= *constant_expr* [*binary_op constant_expr*]

constant_expr ::= [*unary_op*] {*constant_expr* |
INTEGER_CONSTANT}

external_sym_expr ::= [+|-] *EXTERNAL_SYMBOL*

binary_op ::= any legal operator in Chapter 2 that takes two operands.

unary_op ::= any legal operator in Chapter 2 that takes one operand.

where:

EXTERNAL_SYMBOL is a symbol defined with the **.comm**, **.extern**, or **.globl** directives.

INTEGER_CONSTANT is a numeric integer constant value.

When *REL* is a section name or label:

seclab_expr::= [*abs_expr* {+|-}] *sym_expr*

::= *sym_expr* [{+|-} *abs_expr*]

sym_expr ::= [+|-] {*SECTION_NAME* | *LABEL*}

abs_expr is the same as defined above.

where:

SECTION_NAME is a symbol used in the **.sect** directive or any one of the predefined sections.

LABEL is any label.

The use of parentheses is allowed and recommended.

The assembler recognizes and interprets the expression according to where the relocatable expression is found, as shown in Table 5-1.

Table 5-1. Allowed Relocations for Given Fields

Instruction Form/ Data Type	**Absolute/Relative Value**	**Allowed Expression Forms**
I-Form	Absolute branch	VAL2
I-Form	PC-Relative branch	VAL2
B-Form	Absolute branch	VAL2
B-Form	PC-Relative branch	VAL2
D-Form	Absolute	VAL16
.short data directive	Absolute	VAL16[1]

Table 5-1. Allowed Relocations for Given Fields (cont.)

Instruction Form/ Data Type	**Absolute/Relative Value**	**Allowed Expression Forms**
.long data directive	Absolute	VAL

[1] The **SDA** operator is not allowed.

The Instruction Form or Data Type column refers to the type of instruction or data directive that the expression is used in. The Absolute/Relative column refers to whether the expression is considered PC relative or absolute for branch instructions. The Allowed Expression Forms column refers to the expressions that are allowed in that field. The terms VAL2, VAL16, and VAL refer to the grammar specified previously. The PowerPC relocating operators are defined in Chapter 3, "Assembly Language".

Expressions that produce an absolute result are:

REL < *REL*	*REL* != *REL*	*REL* <= *REL*
REL - *REL*	*REL* >= *REL*	*REL* > *REL*
		REL == *REL*

The *REL-REL* expression produces an absolute result only if both *REL* subexpressions are relocatable in the same source program section and defined in the current module (no externals). A relocatable symbol that appears in an expression with any other operator will cause an error (for example, *REL** *REL*).

Note

See Chapter 3, "Assembly Language", for information on valid expressions.

Chapter 6
Assembler Directives

Assembler directives are written as ordinary statements in assembly language. The assembler interprets those statements as commands, which it uses to perform such operations as reserving memory space, defining bytes of data, assigning values to symbols, and controlling the listing output. Assembler directives are case-insensitive.

General Assembler Directives

Table 6-1 lists the general assembler directives by function. Table 6-2 lists the assembler directives alphabetically. Section directives are discussed in Chapter 7, "Section Directives"; macro directives are discussed in Chapter 8, "Macros".

Table 6-1. General Assembler Directives by Function

Directive	Function
Architecture Selection	
.cputype	Sets processor type.
Conditional Assembly	
.else	Specifies an alternate condition.
.elseif	Assembles if value is not zero.
.endif	Ends conditional assembly block.
.if	Assembles if value is not zero.
.ifdef	Assembles if identifier is defined.
.ifeqs	Assembles if strings are equal.
.ifndef	Assembles if identifier is not defined.
.ifins	Assemble if substring occurs within string
.ifnes	Assembles if strings are not equal.
Data Storage Declaration	
.align	Specifies offset alignment.
.aloff	Turns off data directive alignment.
.alon	Turns on data directive alignment.
.byte	Initializes bytes.

Table 6-1. General Assembler Directives by Function (cont.)

Directive	Function
.double	Initializes double-precision values.
.float	Initializes single-precision values.
.long	Initializes words (4 bytes).
.short	Initializes half-words (2 bytes).
.space	Reserves bytes.
.string	Stores strings.
.uadouble	Initializes unaligned double-precision values.
.uafloat	Initializes unaligned single-precision values.
.ualong	Initializes unaligned words (4 bytes).
.uashort	Initializes unaligned half-words (2 bytes).
.vbyte	Initializes one to four bytes.
File Processing	
.end	Ends assembly.
.endian	Sets endianity of object file.
.error	Generates assembly error.
.file	Specifies module name.
.include	Includes source file.
Instruction Controls	
.drop	Unlinks base register expression.
.using	Assigns expression to base register.
.supervisoroff	Indicates the following instructions are in user mode.
.supervisoron	Indicates the following instructions are in supervisor mode.
Listing Control	
.lflags	Sets listing flags.
.list	Enables listing.
.nolist	Disables listing.
.page	Advances to top of page.
.pagelen	Sets maximum page length.
.stitle	Sets listing subtitle.
.title	Sets listing title.

Table 6-1. General Assembler Directives by Function (cont.)

Directive	Function
.width	Sets maximum line width in listing.
.wrapoff	Turns off line wrapping in listing.
.wrapon	Turns on line wrapping in listing.
Macros	
.macro	Begins macro definition.
.mdepth	Sets maximum macro recursion depth.
.mend	Ends macro definition.
.mexit	Terminates macro expansion.
.mexpoff	Turns off macro expansion in listing.
.mexpon	Turns on macro expansion in listing.
.mundef	Purges indicated macros.
Repeat Blocks	
.endrept	Ends repeat block.
.irep	Repeats for each item in list.
.irepc	Repeats for each character in string.
.rept	Repeats *n* times.
Sections (also see Chapter 7)	
.comm	Declares a common symbol.
.dsect	Declares a dummy section.
.org	Sets offset within section.
.pop_sect	Uses a previously saved section.
.push_sect	Saves current section and declares a new section.
.prev	Uses a previously declared section.
.sect	Declares a new section.
Symbol Declaration	
.equ	Assigns a value to a symbol.
.extern	Declares externally visible symbols.
.globl	Declares externally visible symbols.
.rename	Creates alias for symbol.
.set	Assigns a modifiable value to a symbol.

Table 6-1. General Assembler Directives by Function (cont.)

Directive	Function
High-Level	
.merge_end	Indicates end of mergeable region.
.merge_start	Indicates start of mergeable region.

The directives in Table 6-2 without a location reference are described in this chapter.

Table 6-2. Assembler Directives in Alphabetical Order

Directive	Function	Chapter
.align	Specifies offset alignment.	
.aloff	Turns Data Alignment off.	
.alon	Turns Data Alignment on.	
.byte	Initializes bytes.	
.comm	Declare a common symbol.	Chapter 7
.cputype	Sets processor type.	
.double	Initializes double-precision values.	
.drop	Unlinks base register expression.	
.dsect	Declares a dummy section.	Chapter 7
.else	Specifies an alternate condition.	
.elseif	Assembles if value is not zero.	
.end	Ends assembly.	
.endian	Sets endianity of object file.	
.endif	Ends conditional assembly block.	
.endrept	Ends repeat block.	
.equ	Assigns value to a symbol.	
.error	Generates assembly error.	
.extern	Declares symbols as external to this module.	
.file	Specifies file name.	
.float	Initializes single-precision values.	
.globl	Declares symbols as external to this module.	
.if	Assembles if value is not zero.	
.ifdef	Assembles if identifier is defined.	

Table 6-2. Assembler Directives in Alphabetical Order (cont.)

Directive	Function	Chapter
.ifeqs	Assembles if strings are equal.	
.ifins	Assembles if substring occurs within string.	
.ifndef	Assembles if identifier is not defined.	
.ifnes	Assembles if strings are not equal.	
.include	Includes source file.	
.irep	Repeats for each item in list.	
.irepc	Repeats for each character in string.	
.lflags	Sets listing flags.	
.list	Enables listing.	
.long	Initializes word (4 bytes).	
.macro	Begins macro definition.	Chapter 8
.mdepth	Sets maximum macro recursion level.	Chapter 8
.mend	Ends macro definition.	Chapter 8
.merge_end	Indicates end of mergeable region.	
.merge_start	Indicates start of mergeable region.	
.mexit	Terminates macro expansion.	Chapter 8
.mexpoff	Turns off macro expansion in listing.	Chapter 8
.mexpon	Turns on macro expansion in listing.	Chapter 8
.mundef	Purges listed macros.	Chapter 8
.nolist	Disables listing.	
.org	Sets offset within section.	Chapter 7
.page	Advances to top of page.	
.pagelen	Sets maximum page length for listing.	
.pop_sect	Uses a previously saved section.	Chapter 7
.prev	Uses a previously declared section.	Chapter 7
.push_sect	Saves current section and declares a new section.	Chapter 7
.rename	Creates alias for symbol.	
.rept	Repeats *n* times.	
.sect	Declares a new section.	Chapter 7
.set	Assigns modifiable value to a symbol.	

Table 6-2. Assembler Directives in Alphabetical Order (cont.)

Directive	Function	Chapter
.short	Initializes half-words (2 bytes).	
.space	Reserves bytes.	
.stitle	Sets listing subtitle.	
.string	Stores strings.	
.supervisoroff	Indicates the following instructions are in user mode.	
.supervisoron	Indicates the following instructions are in supervisor mode.	
.title	Sets listing title.	
.uadouble	Initializes unaligned double-precision values.	
.uafloat	Initializes unaligned single-precision values.	
.ualong	Initializes unaligned words (4 bytes).	
.uashort	Initializes unaligned half-words (2 bytes).	
.using	Assigns expression to base register.	
.vbyte	Initializes one to four bytes.	
.width	Sets number of columns in listing.	
.wrapoff	Turns line wrapping off.	
.wrapon	Turns line wrapping on.	

Assembler Directives Description

The syntax, a description, and an example of each directive are given in the following pages. The assembler directives are organized alphabetically.

.align — Specifies Offset Alignment

Syntax

.align ***value***

Description

- **value**

  An arithmetic expression that evaluates to a nonzero absolute value at assembly time. The expression may not contain any forward references. The value must be zero or equivalent to some power of two. Zero implies byte alignment.

The **.align** directive specifies that the assembly program counter of the current section be rounded up to the next multiple of *value*. If this value is more restrictive than the alignment of the current section, then the section's alignment is set to *value*.

If the current section is an executable section (that is, it has an executable flag attribute), any padding emitted between the original program counter and the final program counter will be branch instructions that branch to the next instruction. These instructions will only be emitted at the start of a word boundary. Any bytes between the previous program counter and the following word boundary will be filled with zero-valued bytes. If the current section is not an executable section, not a section created with a **.dsect** directive, and not a **bss** section, then all padding between the original program counter and the final program counter will be zero-valued fill bytes.

This directive is normally used following the **.string**, **.space**, **.byte**, and **.short** directives to ensure proper alignment of any directives that follow.

Example

```
.string "odd-sized string."
.align 4
```

In this example, a data directive such as **.long** or **.short** following the **.align** directive will be word-aligned, even though the **.string** directive may leave the program counter at a nonword boundary.

.aloff — Turns Off Automatic Data Alignment

Syntax

.aloff

Description

The **.aloff** directive turns off the automatic alignment that occurs when the **.short**, **.long**, **.float**, and **.double** directives are used. When the alignment feature is turned on, padding will be emitted by the assembler if the assembly program counter is not on a boundary that is desirable for that data directive. This directive can be used to avoid this additional padding.

The **.aloff** directive will set the **ALDATA** assembler symbol value to zero (0).

The default setting for the assembler is to perform automatic alignment.

Example

```
.aloff
.align 4
.byte 1
.double 3.0
```

In this example, the double floating point-value will not be automatically aligned to an a boundary appropriate for the processor the data is being assembled for. Instead, the first byte of the double will be emitted immediately after the byte value. Unaligned data can require additional time for accesses to that data.

.alon — Turns on Automatic Data Alignment

Syntax

.alon

Description

The **.alon** directive turns on the automatic alignment that occurs when the **.short**, **.long**, **.float**, and **.double** directives are used. When the alignment feature is turned on, padding will be emitted by the assembler if the assembly program counter is not on a boundary that is desirable for that data directive. This is the default setting for the assembler.

The **.alon** directive will set the **ALDATA** assembler symbol value to one (1).

Example

```
.alon
.align 4
.byte 1
.double 3.0
```

In this example, the double floating-point value will not immediately follow the byte value. Instead, there will be enough additional padding emitted to bring the assembly program counter to a four-byte or eight-byte boundary, depending on the processor the data is being assembled for. This alignment is most beneficial for double floating-point accesses.

.byte — Initializes Bytes

Syntax

.byte [*n*] ***value*** [, [*n*] *value*]...

Description

- **[*n*] value**

  A list of arbitrary expressions separated by commas (**,**). Each expression can be either an absolute value or a string. The expression can be duplicated *n* times by specifying the repeat value within square brackets (**[]**) before the expression that is to be repeated. The repeat value, if present, must be a non-forward-referencing absolute value. If the repeat value and its surrounding square brackets are not used, then a single copy of the expression is emitted.

The **.byte** directive generates initialized bytes containing the eight-bit values represented in *value*. If the value cannot be represented in a single byte, it is truncated on the left. If the value is a string, the string is stored in byte order. The string will not be terminated by a final zero byte (that is done by the **.string** directive). Alignment with the **.align** directive should be considered after using the **.byte** directive because the assembly program counter can be left at an undesirable boundary.

Example

```
.set eight_bits,12
.byte [2]1,eight_bits,5+5
.align 8
```

In this example, the **.byte** directive generates a byte stream of 1,1,12,10 and is aligned on the nearest eight-byte boundary. Specifically, **[2]1** generates two 1s; **eight_bits** generates **12** because the symbol is set to that value; and **5+5** generates **10**.

.cputype — Sets Processor Type

Syntax

.cputype "*processor*[*modifier*]"

Description

- **processor**

  Processor type. The processor type can be one of the following: **401gf, 403ga, 403gb, 403gc, 403gcx, 405cr, 405ep, 405gp, 405gpr, 440ep, 440gp, 440gx, 505, 509, 5121e, 5123, 5200, 5200b, 533, 534, 535, 536, 555, 5553, 5554, 556, 560, 561, 562, 563, 564, 565, 566, 5xx, 603, 603e, 603e6, 603e7v, 603r, 604, 604e, 740, 7400, 7410, 7410t, 7441, 7445, 7445a, 7447, 7447a, 7448, 745, 7450, 7451, 7455, 7455a, 7457, 745b, 750, 750cx, 750cxe, 750cxr, 750fl, 750fx, 750gl, 750gx, 750gx_dd11_8, 750l, 755, 755b, 755c, 801, 821, 823, 823e, 8240, 8241, 8245, 8245a, 8247, 8248, 8250, 8250a, 8255, 8255a, 8260, 8260a, 8264, 8264a, 8265, 8265a, 8266, 8266a, 8270, 8270vr, 8271, 8272, 8275, 8275vr, 8280, 8313, 8313e, 8314, 8314e, 8315, 8315e, 8321, 8321e, 8323, 8323e, 8343, 8343e, 8347, 8347e, 8349, 8349e, 8358e, 8360e, 8377e, 8378e, 8379e, 850, 850de, 850dsl, 850sr, 852t, 853t, 8533, 8533e, 8540, 8541e, 8543, 8543e, 8544, 8544e, 8545, 8545e, 8547e, 8548, 8548e, 8555e, 855t, 8560, 8567, 8567e, 8568, 8568e, 8572, 8572e, 857dsl, 857t, 859dsl, 859p, 859t, 860, 860de, 860dp, 860dt, 860en, 860p, 860sr, 860t, 8610, 862, 862p, 862t, 8640, 8640d, 8641, 8641d, 866, 866p, 866t, 870, 875, 880, 885, 8xx, com, e300, e300c2, e300c3, e500v1, e500v2, ec603e, ec603e6, ec603e7v, em603e, g2, g2_le, npe405h, npe405l,** and **ppc**.

The default processor type is "**603**."

- ***modifier***

  Indicates whether floating-point instructions should be allowed. If all floating-point instructions are to be accepted without warning, the **/F** modifier can be added immediately after the processor type. If those instructions are not to be allowed, the **/NF** modifier can be used. If neither modifier is used, the assembler will allow any instructions that are executable directly on the indicated processor’s hardware.

The **.cputype** directive specifies the target processor on which the object file is meant to be executed. If an instruction mnemonic or special purpose register is found that will not execute on the indicated processor, a warning is displayed to indicate that there may be problems executing this code.

The **.cputype** directive, if present, must be used before any instructions are processed.

While some processors, such as the 603 and 604, support all of the floating-point instructions, other processors, such as the 860 and EC603E, do not. For example, any attempt to execute floating-point instructions on the 860 will result in a run-time exception. This can be a problem unless software simulation is available for those instructions that caused the exception. If a software simulation routine is available, use of instructions outside of the processor’s normal set of instructions is allowed. This support is indicated to the assembler through the use of the **/F** modifier to the processor type. The linker will then verify that software simulation is linked into the final executable so the program will execute correctly.

An alternative use is to use the **/NF** modifier to remove the floating-point instructions from the set of instructions accepted by the assembler for a given processor. This may be useful if the "floating-point available" bit is not set in the MSR. This modifier cannot be used to make more instructions available; it only removes instructions from the set of available assembly instructions.

Example

```
    .cputype "603"
```

In this example, the target processor is specified as the 603 processor.

.double, .uadouble— Initializes Double-Precision Values

Syntax

.double [*n*] ***real_constant*** [, [*n*] *real_constant*]...

.uadouble [*n*] ***real_constant*** [, [*n*] *real_constant*]...

Description

- **[*n*] real_constant**

  A list of real constants separated by commas. Each *real_constant* can be duplicated *n* times by specifying the value within square brackets ([]) before the *real_constant* that is to be repeated. The repeat value, if present, must be a nonforward-referencing absolute value. If the repeat value and its surrounding square brackets are not used, then a single copy of the expression is generated.

The **.double** and **.uadouble** directives generate 64-bit double-precision floating-point numbers in IEEE format. No relocatable expressions are allowed. See Chapter 3, “Assembly Language”, for descriptions of how to specify real constants.

If the assembly program counter is not on an appropriate boundary for the **.double** directive, and if data alignment is turned on (as indicated by the **ALDATA** assembler symbol), the assembler program counter will be aligned before the data is emitted. The amount of alignment depends on the processor that the data is being aligned for. This alignment may be four-byte or eight-byte, depending on the processor’s data bus size. Any label on the same line as the directive will have its offset value adjusted to the aligned boundary.

The **.uadouble** directive never checks for data alignment.

Example

```
.double [2*3]1,3.0
```

This is equivalent to:

```
.double 1.0,1.0,1.0,1.0,1.0,1.0,3.0
```

In this example, seven floating-point numbers are generated. The expression **[2*3]1** generates six **1.0**s, followed by **3.0**.

.drop — Unlinks Expression from Base Register

Syntax

.drop ***value***

Description

- **value**

  An expression that evaluates to an absolute value at assembly time. This value can be a general register symbol (such as **r3**) or an absolute value that is between 0 and 31, inclusive. The expression cannot contain any forward references.

The **.drop** directive removes the association of a general register and an expression. The indicated base register will no longer be considered when processing displacements for instructions that do not have an indicated base register. No error is reported if the specified base register does not have an expression currently assigned to it.

This directive does not affect the contents of the general register in any way. It only affects whether or not the assembler will be able to automatically determine which base register to use for an instruction that does not specify a base register but requires one. Use of the **.drop** directive will prevent the assembler from considering the indicated base register until the base register is reassigned a new value through the **.using** directive.

Example

```
.dsect dummy
d1:
.using dummy,r5
lhau r3,d1# Assuming d1 is a label in a dsect
         # section
.drop r5
lhau r3,d1# error, no matching .USING register
```

In this example, the general register **r5** is assumed to contain the address of the **dsect** section. When a label from that section is used in an instruction with no base register indicated, the assembler will supply the missing base register. However, once the association between the dummy **dsect** section and the general register is cleared, the assembler will be unable to determine what base register should be used for the instruction and will emit an error.

.else — Specifies an Alternate Condition

Syntax

.else

Description

The **.else** directive specifies the start of an alternative clause. If the condition part of the corresponding **.if** directive evaluates to false, only the statements following the **.else** directive will be assembled and written to the output file. Conditional assembly terminates with a **.endif** directive.

The **.if** directives are **.if**, **.ifdef**, **.ifeqs**, **.ifins**, **.ifndef**, **.ifnes**,

and .elseif'.

The **.else** directive is optional and can only appear once for each **.if** directive prior to the associated **.endif** directive.

This directive may not be used in conjunction with a **.if** directive from a different assembly file, such as through the use of the **.include** directive.

Example

```
.if 2>3
.long 1
.else
.long 2
.endif
```

In this example, the statement **.long 2** is assembled because the **.if** directive evaluates to false.

.elseif — Assembles if Value Is Not Zero

Syntax

.elseif ***expression***

Description

- **expression**

  An expression that evaluates to an absolute value at assembly time. There can be no forward references in the expression.

The **.elseif** directive must be contained between the **.if** and **.endif** directives. If the matching **.if** directive or a preceding **.elseif** directive has evaluated to be true, then a **.elseif** directive is not evaluated and any statements between the **.elseif** and its terminating **.elseif**, **.else**, or **.endif** directive are ignored. If, however, the matching **.if** directive and any preceding **.elseif** directive have evaluated to false conditions, then the expression associated with the **.elseif** directive is evaluated. If the resulting expression is nonzero, then any statements between the **.elseif** and its terminating **.elseif**, **.else**, or **.endif** directive are evaluated. If the result of the expression is zero, the statements are ignored.

Note that all comparisons are signed.

This directive cannot be used without a preceding **.if** directive and must be terminated with either another **.elseif**, **.else**, or **.endif** directive. Any **.if** directives associated with a **.elseif** directive must be contained within the same source file.

Example

```
.if N>100
.error "*** ERROR: N is too big"
.elseif N > 50
.string "This value is greater than 50"
.long N
.elseif N > 25
.string "This value is between 25 and 50"
.long N
.else
.string "This value is less than or equal to 25"
.long N
.endif
```

In this example, if **N** is greater than **100**, the error message ***** ERROR: N is too big** is written to standard output and an error condition is signalled. If the value is greater than 50 but less than or equal to 100, the string and value in the second clause are evaluated. If the value is greater than 25 but less than or equal to 50, the string and value in the third clause are evaluated. If the value is less than or equal to 25, the fourth clause is evaluated.

.end — Ends Assembly

Syntax

```
.end
```

Description

The **.end** directive terminates the assembly of instructions. All text beyond the **.end** is ignored. The assembler issues a warning if nonwhite space (that is, a character other than blank, tab, or newline) follows **.end**.

.endian — Sets Endianity of Object File

Syntax

.endian {**big** | **little**}

Description

- **big**

  Generates big-endian byte order in the object file.

- **little**

  Generates little-endian byte order in the object file.

The **.endian** directive specifies the byte order in the object module generated by the assembler. It also affects the byte ordering within the listing.

If there is a conflict between the **-E** command line option and the **.endian** assembler directive, an error is emitted and the command line option is used. The **.endian** directive may only be used once in an assembly source file.

The default endianity is **big**.

Example

```
    .endian big
```

In this example, the assembler generates an object module with big-endian byte order.

.endif — Ends Conditional Assembly Block

Syntax

.endif

Description

The **.endif** directive terminates conditional assembly. In the case of nested **.if** directives, a **.endif** is paired with the most recent **.if** directive.

The **.if** directives are **.if**, **.ifdef**, **.ifeqs**, **.ifins**, **.ifndef**, **.ifnes**, and **.elseif**.

The **.endif** directive must be in the same assembly file as its matching **.if** directive. It cannot be used in conjunction with a **.if** directive from a different assembly file, such as through the use of the **.include** directive.

Example

```
.set sum,0
.if sum == 4
ADDI R4,R5,sum
.set sum,sum+1
.else
ADDI R4,R5,sum+5
.endif
```

In this example, statements following **.if** and preceding the **.else** directive are assembled if **sum==4** is not zero. Otherwise, statements between the **.else** and **.endif** directives are assembled.

.endrept — Ends Repeat Block

Syntax

.endrept

Description

The **.endrept** directive terminates a repeat block defined by the **.rept**, **.irep**, or **.irepc** directives. Note that the **.endrept** directive does not terminate a macro definition.

Example

```
.rept 10
.byte 1
.endrept
```

This example is equivalent to:

```
.byte 1
.byte 1
.byte 1
.byte 1
.byte 1
.byte 1
.byte 1
.byte 1
.byte 1
.byte 1
```

In this example, ten bytes of **1**s are generated.

.equ — Assigns a Value to a Symbol

Syntax

.equ ***symbol***,***expression***

Description

- **symbol**

  A symbol.

- **expression**

  An arbitrary absolute or relocatable expression whose value will be assigned to the symbol. The expression can contain forward references. It is not valid to assign strings, floating-point constants, or complex relocation expressions.

The **.equ** directive sets a symbol equal to a particular value. The symbol cannot already be defined except as part of a **.globl** or **.extern** directive. The symbol can be used anywhere that its assigned expression could be used, even in forward-references in instructions or directives before the **.equ** directive is defined.

Only a single **.equ** directive for a given symbol can exist in a source program. The symbol will be emitted into the object file. If the symbol was globally defined with the **.extern** or **.globl** directive, the symbol can also be used in other source programs. Otherwise, the symbol is only available within the source program where it is defined.

If a symbol is needed whose value needs to be modified during assembly, the **.set** directive should be used.

Example

```
.long n
.equ n,1
.long n
```

The above code is equivalent to the following statements:

```
.long 1
.long 1
```

.error — Generates Assembly Error

Syntax

.error ***text***

Description

- **text**

  A message to be displayed.

The **.error** directive indicates an error condition in the listing and error summary. This directive is typically used within macros or conditional assembly to flag user-defined error conditions.

When a **.error** directive is assembled, the assembler marks it with an error message. The text within the string is printed to the terminal and listing (if emitted), normally stating the reason for the error. The assembler immediately stops processing of the text file.

Example

```
.if size>MAXSIZE
.error "ERROR: size is too big"
.endif
```

In this example, the error message **ERROR: size is too big** is printed and an error condition is flagged if **size** exceeds **MAXSIZE**. Any statements following this statement will not be processed.

.extern, .globl — Declares Symbols as Visible External to This Module

Syntax

.extern ***symbol*** [, *symbol*]...

.globl ***symbol*** [, *symbol*]...

Description

- **symbol**

  An identifier representing a relocatable value.

The **.extern** and **.globl** directives specify a list of symbol names that can be referenced from other modules. This directive can appear anywhere in the source file. Use of either directive results in the same behavior.

If the symbol used with this directive is defined within the current module as a label or through the use of the **.comm** or **.equ** directive, then the symbol will be made globally visible. Otherwise, it is assumed that this symbol is not defined within this module and is expected to be defined in another module.

It is not valid to make a symbol created with the **.set** directive globally defined, due to the possible redefinition of the symbol during assembly. Instead, either the **.equ** directive should be used, which can only be assigned a value once during assembly, or a global absolute value can be set at link time through the **PUBLIC** linker command.

Symbol names can be declared as externally defined any number of times. If an undefined symbol is encountered in a relocatable expression, the assembler will issue an undefined symbol warning message and treat the symbol as though it had appeared in a **.extern** directive.

Example

```
 .extern _printf,_flags,myfunc
 .comm _flags, 5
myfunc:
```

In this example, **_printf** would be externally defined while **_flags** and **myfunc** would be made globally defined for use by another module.

.file — Specifies Filename

Syntax

.file ***text***

Description

- **text**

  A string or text that does not contain white space. The value of *text* is commonly used to identify the source name of the program being assembled. Any backslash character (\) within the text is taken as-is and is not used as an escape character.

The **.file** directive specifies the source name of the program being assembled. There are no restrictions on *text* if the text is enclosed within quotes. Without quotes, the text cannot contain any white space. The contents of *text* will be placed into the symbol table of the object. This symbol may be used in a future release for debugging purposes.

This directive can only appear once in an assembly module. Subsequent occurrences have no effect. If this directive is not used in a module, the name of the source file, without any path information, will be placed in the symbol table of the object.

Example

```
    .file sieve.c
```

In this example, the file symbol **sieve.c** will be placed in the symbol table of the object file.

.float, .uafloat — Initializes Single-Precision Values

Syntax

.float [*n*] ***real_constant*** [, [*n*] *real_constant*]...

.uafloat [*n*] ***real_constant*** [, [*n*] *real_constant*]...

Description

- **[n] real_constant**

  A list of real constants separated by commas (**,**). Each *real_constant* can be duplicated *n* times by specifying the value within square brackets (**[]**) before the *real_constant* that is to be repeated. The repeat value, if present, must be a nonforward-referencing absolute value. If the repeat value and its surrounding square brackets are not used, then a single copy of the expression is emitted.

The **.float** and **.uafloat** directives generate 32-bit floating-point numbers in IEEE format. No relocatable values are allowed. See Chapter 3, "Assembly Language", for descriptions of how to specify real constants.

For the **.float** directive, if the assembly program counter is not on a 4-byte boundary and if data alignment is turned on (as indicated by the ALDATA assembler symbol), the assembler program counter will be aligned to a four-byte boundary before the data is emitted. Any label on the same line as the directive will have its offset value adjusted to the aligned boundary.

The **.uafloat** directive never checks for data alignment.

Example

```
.float [3]1.0,2
```

This is equivalent to:

```
.float 1.0,1.0,1.0,2.0
```

In this example, four floating-point numbers are generated. Three **1.0**s are generated first, followed by a **2.0**.

.if — Assembles if Argument is Not Zero

Syntax

.if ***expression***

Description

- **expression**

  An expression that evaluates to an absolute value at assembly time. Forward references are not allowed in the expression.

The **.if** directive specifies that all source text following the **.if** statement and preceding its corresponding **.elseif**, **.else**, or **.endif** directive be assembled if the expression evaluates to nonzero at assembly time.

Note that all comparisons are signed.

This directive can be nested within other conditional assembly directives. The maximum nesting level is 15.

This directive and its corresponding **.elseif**, **.else**, and **.endif** directives must be in the same assembly source file.

Example

```
.if N>100
.error "*** ERROR: N is too big"
.endif
```

In this example, if **N** is greater than **100**, the error message ***** ERROR: N is too big** is written to standard output and an error condition is signalled.

.ifdef — Assembles if Identifier is Defined

Syntax

.ifdef ***identifier***

Description

- **identifier**

  An identifier.

The **.ifdef** directive specifies that all code following the **.ifdef** statement be assembled if *identifier* is already known by the assembler. Conditional assembly terminates with a corresponding **.else** or **.endif** directive.

A typical use for this directive is in conjunction with the **-D** assembler command line option (see the *User's Guide* section of this manual). This option defines a symbol and equates a value to this symbol. The **.ifdef** directive can be used to determine whether this symbol has been defined with the **-D** option.

This directive can be nested within other conditional assembly directives. The maximum nesting limit is 15.

This directive and its corresponding **.else** and **.endif** directives must be in the same assembly source file.

Example

```
.ifdef TABSIZE
.space TABSIZE
.endif
```

In this example, **TABSIZE** bytes are allocated if **TABSIZE** has been defined.

.ifeqs — Assembles if Strings Are Equal

Syntax

.ifeqs "***string1***","***string2***"

Description

- **string1, string2**

  Any text.

The **.ifeqs** directive specifies that all code following the **.ifeqs** statement can be assembled if *string1* is identical to *string2*. Conditional assembly terminates with a corresponding **.else** directive or a **.endif** statement.

The string can also be delimited by the single quote character (‘) instead of the double quote character (“). Macro parameter substitution occurs within single quotes, but not double quotes.

If a string is assigned to a symbol with the **.set** directive, that symbol can be used instead of a quoted string.

This directive is most useful inside macros to test the macro parameters. For more information on macros, see Chapter 8, “Macros”.

The **.ifeqs** directive can be nested within other conditional assembly directives. The maximum nesting limit is 15.

This directive and its corresponding **.else** and **.endif** directives must be in the same assembly source file.

Example

```
.macro data,val,type
.ifeqs ‘type’,"int"
.long val
.else
.ifeqs ‘type’,"float"
.float val
.endif
.endif
.mend

data 100,int
```

In this example, the statement **.long 100** would be assembled.

.ifins — Assembles if Substring Occurs Within String

Syntax

.ifins "***substring***","***string***"

Description

- **substring, string**

  Any text.

The **.ifins** directive specifies that all code following the **.ifins** statement can be assembled if *substring* occurs within *string*. Conditional assembly terminates with a corresponding **.else** directive or a **.endif** statement.

If a string is assigned to a symbol with the **.set** directive, that symbol can be used instead of a quoted string.

This directive is most useful inside macros to test the macro parameters. For more information on macros, see Chapter 8, “Macros”.

The string can also be delimited by the single quote character (‘) instead of the double quote character (“). Macro parameter substitution occurs within single quotes, but not double quotes.

The **.ifins** directive can be nested within other conditional assembly directives. The maximum nesting limit is 15.

This directive and its corresponding **.else** and **.endif** directives must be in the same assembly source file.

Example

```
.macro move,src,dest
 .ifins "(",'src'
 lwz dest,src
 .else
 .ifins "(",'dest'
 stw src,dest
 .else
 .ifnes 'src','dest'
 ori dest,src,0
 .endif
 .endif
 .endif
 .mend
 move r3,16(r1)
```

In this example, the instruction stw r3,16(r1) would be assembled.

.ifndef — Assembles if Identifier is Not Defined

Syntax

.ifndef ***identifier***

Description

- **identifier**

  An identifier.

The **.ifndef** directive specifies that all code following the **.ifndef** statement can be assembled if *identifier* is not already known by the assembler. Specifically, if *identifier* exists in the symbol table, the condition fails, and the code is not assembled. Conditional assembly terminates with a corresponding **.else** or **.endif** directive. This directive will not find any of the predefined register symbols, although it will match predefined section names.

A typical use for this directive is in conjunction with the **-D** assembler command line option (see the *User's Guide*). This option defines a symbol and equates a value to this symbol. The **.ifndef** directive can be used to determine whether this symbol has been defined with the **-D** option.

The **.ifndef** directive can be nested within other conditional assembly directives. The maximum nesting limit is 15.

This directive and its corresponding **.else** and **.endif** directives must be in the same assembly source files.

Example

```
.ifndef TABSIZE
.set TABSIZE,100
.endif
```

In this example, **TABSIZE** is defined to be **100** if it is not previously defined.

.ifnes — Assembles if Strings Are Not Equal

Syntax

.ifnes "***string1***","***string2***"

Description

- **string1, string2**

  Any text.

The **.ifnes** directive specifies that all code following the **.ifnes** statement can be assembled if *string1* is not the same as *string2*. Only one character needs to be different between these two strings for the directive to evaluate to true. Conditional assembly terminates with a corresponding **.else** or **.endif** directive.

The string can also be delimited by the single quote character (‘) instead of the double quote character (“). Macro parameter substitution occurs within single quotes, but not double quotes.

If a string is assigned to a symbol with the **.set** directive, that symbol can be used instead of a quoted string.

This directive is most useful inside macros to test the macro parameters. For more information on macros, see Chapter 8, “Macros”.

The **.ifnes** directive can be nested within other conditional assembly directives. The maximum nesting limit is 15.

This directive and its corresponding **.else** and **.endif** directives must be in the same assembly source file.

Example

```
.macro nulltest,string
.ifnes string,""
.string "string is not null"
.endif
.endm
```

In this example, the character string **string is not null** is generated if the **string** parameter to the macro named **nulltest** contains at least one character.

.include — Includes Source File

Syntax

.include "[*pathname*]***filename***"

Description

- ***pathname***

  The search path leading to the file specified.

- **filename**

  The name of a file to be included for assembly.

The **.include** directive inserts an external source file into the input source code stream at assembly time. The **.include** statements can be nested and can include macro calls. A macro call can also contain a **.include** directive. The maximum number of nested include files is host dependent, but it can also be modified using the **.mdepth** directive.

If additional include directories are passed in on the command line, those directories are searched if the file is not found in the current directory. If the source file cannot be found at all, an error message is printed and assembly continues. Searching directories is not required if a full path is specified.

The optional *pathname* specification and the *filename* are passed to the host operating system as received, without any lower-case to upper-case conversion. Likewise, any backslash characters (\) within the string are used as-is and are not considered to be escape characters.

If a **.include** directive is within a false clause of a **.if** directive, the directive will not be processed.

Example

```
.include "external.src"
```

In this example, the contents of the external source file **external.src** is inserted into the input source code at the point where **.include** is encountered in the assembly source file.

.irep — Repeats for Each Item in List

Syntax

.irep ***identifier*** [,*list*]

Description

- **identifier**

  A symbol which will take on a different value for each iteration of the repeated text.

- ***list***

  An optional list of arbitrary character sequences separated by commas. If no list is given, zero repetitions will occur.

The **.irep** directive assembles statements up to the next **.endrept** one time for each element of *list*. Each element in *list* is separated from the other by a comma (**,**). Occurrences of *identifier* within text are replaced by an element from the *list*. As with macros, the angle bracket (<>) characters can be used to separate *identifier* from surrounding text when *identifier* is embedded in alphanumeric text. The angle brackets are removed after the text is replaced.

Repeat directives can be contained within conditional code, macro definitions, or other repeat directives.

Since the comma is used as a separator for arguments, it may only be used as part of an argument outside of a string by putting a backslash before the comma character. The comma will become part of the argument and the backslash will be removed. It should be noted that any backslash character outside of a string in an argument will be removed while the character following the backslash (which might be another backslash) will be retained as part of the argument.

Example

```
        .irep par,1,2,4,10
a<>par:.byte par
        .endrept
```

In this example, the following statements are generated:

```
a1:  .byte 1
a2:  .byte 2
a4:  .byte 4
a10: .byte 10
```

.irepc — Repeats for Each Character in String

Syntax

.irepc ***identifier***,"***string***"

Description

- **identifier**

  A symbol which will take on a different value for each iteration of the repeated text.

- **string**

  Any text. If *string* is empty, zero repetitions will occur.

The **.irepc** directive assembles statements up to the next **.endrept** directive one time for each character in *string*. Occurrences of *identifier* within text are replaced by a character of *string*. As with macros, the angle bracket (<>) characters can be used to separate *identifier* from surrounding text when *identifier* is embedded in alphanumeric text. The angle brackets are removed after the text is replaced.

Repeat directives can be contained within conditional code, macro definitions, or other repeat directives.

Example

```
         .irepc p,"ABC"
char_<>p:.byte ’p
         .endrept
```

In this example, the following statements are generated:

```
char_A: .byte 'A
char_B: .byte 'B
char_C: .byte 'C
```

.lflags — Sets Listing Flags

Syntax

.lflags ***flag***[*flag*]...

Description

- **flag**

  Any of the assembler command line flags listed in the section Assembler Command Flags in Chapter 2, Command Usage.

The **.lflags** directive controls features of the assembler behavior like the **flags** command line option, but from the source text instead of the command line. The format and function of the flags are identical to those specified by the **flags** command line option.

The **n** character may be placed in front of any flag to negate the following flag. Only the flag immediately following the **n** character is affected by the negation. If more than one flag is to be negated, multiple **n** characters must be used.

The **a**, **c**, **d**, **i**, **m**, **o**, and **w** flags become effective immediately following the **.lflags** directive and remain in effect until assembly completes or until a subsequent **.lflags** directive resets the flag.

The **b**, **f**, **k**, **l**, **p**, **r**, **s**, **v**, and **x** flags may only be set once. Any attempt to reuse the flag will result in an error. If one of these flags is used on the command line, the flag's setting cannot be overridden in the assembly source file.

The command line **flags** option is functionally equivalent to placing a **.lflags** directive at the beginning of the source file.

The **.lflags** directive must be used before any symbols have been created or referenced. Otherwise, early uses of those symbols will not appear in the cross reference of the listing. The symbols themselves are still available to other directives or instructions within the assembly source file.

Commas or white space can be used to separate flags, if desired. They are ignored in that context.

Example

```
.lflags ina    # turn on i flag, turn off a flag
  ...
.lflags nia    # turn off i flag, turn on a flag
```

The **i** flag indicates that **.include** files are to be listed. The **a** flag indicates that the output of data directives is to be automatically aligned to the appropriate boundary, as needed. In this example, **.include** files occurring between the two **.lflags** directives are listed. They are not listed elsewhere (assuming the flag was not used outside of this example). Likewise, data directives between the two **.lflags** directives will not be aligned to appropriate boundaries. Outside of these directives, automatic alignment will occur.

.list — Enables Listing

Syntax

.list

Description

The **.list** directive causes assembly lines to be emitted to the assembly listing. This continues until either the end of the assembly listing or until a **.nolist** directive is seen. If not specified, the **.list** directive is on by default.

This directive is most useful when used in conjunction with the **.nolist** directive to selectively list portions of an assembler listing. If the listing command line option (**-l**) is not used, this directive is ignored because no assembler listing is produced.

.long, .ualong — Initializes Words

Syntax

.long [*n*] ***expression*** [, [*n*] *expression*]...

.ualong [*n*] ***expression*** [, [*n*] *expression*]...

Description

- **[n] expression**

  An arbitrary expression. Each *expression* can be duplicated *n* times by specifying the repeat value within square brackets (**[ ]**) before the *expression* that is to be repeated. The repeat value, if present, must be a nonforward-referencing absolute value. If the repeat value and its surrounding square brackets are not used, then a single copy of the expression is emitted.

The **.long** and **.ualong** directives generate initialized words containing the 32-bit values represented in *expression*. Absolute values that will not fit within four bytes are truncated on the left. Any relocatable value may fit within this directive. If the expression is a 16-bit relocatable value, it will be emitted in the lower two bytes of the data directive, depending upon the endianity setting.

For the **.long** directive, if the assembly program counter is not on a four-byte boundary and if data alignment is turned on (as indicated by the **ALDATA** assembler symbol), the assembler program counter will be aligned to a four-byte boundary before the data is emitted. Any label on the same line as the directive will have its offset value adjusted to the aligned boundary.

The **.ualong** directive never checks for data alignment.

Example

```
.long $+4
```

In this example, the 32-bit value of the current program counter plus the arithmetic value of **4** is generated.

```
.long [4]1
```

In this example, the following is generated: 0x00000001, 0x00000001, 0x00000001, and 0x00000001.

.merge_end — Indicates End of Mergeable Region

Syntax

.merge_end

Description

The **.merge_end** directive indicates the end of a mergeable region of code or data. There must be a preceding **.merge_start** directive.

Notes

This directive is used for C++ compiler support. Use of this directive in manually written code is not recommended.

.merge_start — Indicates Start of Mergeable Region

Syntax

.merge_start ***symbol*** [, *letter* [, *letter*] ...]

Description

- **symbol**

  An identifier representing a label within the mergeable region.

- ***letter***

  A single letter indicating the type of mergeable region.

The **.merge_start** directive is used to indicate a mergeable region of code or data. The **.merge_start** directive must be followed by a **.merge_end** directive. It is expected that the symbol used with this directive will exist within the mergeable region. A warning will be emitted if the label is not at the beginning of the mergeable region.

References to symbols within a mergeable region will require the use of relocation expressions. This is necessary since the copy of the mergeable region that is actually retained in the executable may come from a different module.

Notes

This directive is used for C++ compiler support. Use of this directive in manually written code is not recommended.

.nolist — Disables Listing

Syntax

.nolist

Description

The **.nolist** directive suppresses the output of assembly lines to the assembly listing, except for those lines flagged with errors. This suppression continues until either the end of the assembly listing or until a **.list** directive is seen.

This directive is most useful when used in conjunction with the **.list** directive to selectively list portions of an assembler listing. If the listing command line option (**-l**) is not used, this directive is ignored because no assembler listing is produced.

.page — Advances to Top of Page

Syntax

```
.page
```

Description

The **.page** directive skips to the top of the next page on the listing form. The readability of source programs is greatly improved when each subroutine begins on a new page.

A header is printed at the top of each page. The title and subtitle strings printed as part of the header are defined through the use of the **.title** and **.stitle** directives. These strings can also be accessed through the use of the **LTITLE** and **LSTITLE** assembler symbols.

If the listing command line option (**-l**) is not used, this directive is ignored because no assembler listing is produced.

.pagelen — Sets Maximum Page Length in Listing

Syntax

.pagelen ***value***

Description

- **value**

  An expression that evaluates to an absolute value at assembly time. This value cannot contain any forward references. This value may be 0 or a value greater than 20.

The **.pagelen** directive sets the maximum page length for the listing. An automatic page eject will occur if the specified number of lines have been added to the list since the last page eject. This line count will include the header, error, and warning messages. However, no page eject will occur after the initial header if the page length is set to 0.

This directive is the same as using the **p** flag in the **.lflags** directive.

The **.pagelen** directive sets the **LPAGE** assembler symbol value to the indicated value.

The default page length is 55.

If the listing command line option (**-l**) is not used, this directive is ignored because no assembler listing is produced.

Example

```
.pagelen 40
```

In this example, the page length is set to 40. An automatic page eject will occur whenever 40 lines of text are processed after this directive is executed.

.rename — Creates an Alias for a Symbol

Syntax

.rename ***symbol***, ***string***

Description

- **symbol**

  A symbol that is defined elsewhere within the assembly source file. This symbol can be a label, external symbol, common symbol, and so forth. It cannot be a symbol created through the use of the **.set** directive or a section directive.

- **string**

  A string containing the new name for the symbol.

The **.rename** directive is used to create a link between two names. The first name is a symbol that is defined within the assembly source file and possibly used in other directives or instructions. The second name is a potentially invalid symbol name that cannot be defined directly within the assembly file. The assembler will emit the second name into the symbol table of the object file in place of the first name.

This directive can be used to create symbol names that cannot be created under the limitations of the assembler’s syntax. It is not possible to rename a symbol more than once or to rename a symbol to a name that has already been defined either within the module or as predefined by the assembler.

Example

```
.rename internsym,"Illegal Sym"
.extern internsym
internsym:
```

In this example, the alias is created between the symbol **internsym** and the string **Illegal Sym**. The symbol is used within the assembly source file. However, the symbol table entry in the object file will contain the text within the string.

.rept — Repeats N Times

Syntax

.rept ***n***

Description

- **n**

  An arbitrary expression that evaluates to an absolute value. This expression cannot contain forward references.

The **.rept** directive repeats an assembly block a specified number of times. The statements to be repeated are those between **.rept** and the following **.endrept** directive. The statements are expanded from the point at which the **.rept** directive is encountered.

Repeat directives can be contained within conditional code, macro definitions, or other repeat directives.

Example

```
    .rept 4
L!: ADD r1,r2,r3
   bc 0xC,1,L!
    NOP
    .endrept
```

In this example, any statements between the **.rept** directive and the following **.endrept** directive are repeated four times. In each iteration of the repeat block, the exclamation mark (**!**) is replaced by an underscore (_) and the current value of the **BCOUNT** assembly variable. This variable is incremented once for each iteration of the repeat block, so the result is the creation of four unique labels. If the **BCOUNT** assembly variable had a value of two before this repeat block was executed, the resulting output would be:

```
L_3: ADD r1,r2,r3
   BC 0xC,1,L_3
    NOP
L_4: ADD r1,r2,r3
   BC 0xC,1,L_4
    NOP
L_5: ADD r1,r2,r3
   BC 0xC,1,L_5
    NOP
L_6: ADD r1,r2,r3
   BC 0xC,1,L_6
    NOP
```

.set — Assigns a Modifiable Value to a Symbol

Syntax

.set ***symbol*,*expression***

Description

- **symbol**

  A symbol.

- **expression**

  An arbitrary absolute or relocatable expression whose value will be assigned to *symbol* until changed by another **.set** directive. The expression cannot contain any forward references. The expression can also be a floating-point constant or a string.

The **.set** directive sets a symbol equal to a particular value. The symbol cannot already exist unless it was created through an earlier **.set** directive. This symbol can be used anywhere that its assigned expression, floating-point value, or string could be used.

A source program can have multiple **.set** directives for the same symbol. The most recent **.set** directive determines the value of the symbol until another **.set** directive is processed. The symbol used with a **.set** directive cannot be forward referenced. Also, the symbol cannot be globally defined with the **.extern** or **.globl** directives.

If an expression is needed in multiple modules, either the **.equ** directive should be used to define the value or the **PUBLIC** command in the linker command file should be used to assign the expression to a global symbol. Both methods will produce equivalent results.

Example

```
.set n,1
.rept 4
.byte n
.set n,n+n
.endrept
```

The above code generates the following statements:

```
.byte 1
.byte 2
.byte 4
.byte 8
```

.short, .uashort — Initializes Half-Words

Syntax

.short [*n*] ***expression*** [, [*n*] *expression*]...

.uashort [*n*] ***expression*** [, [*n*] *expression*]...

Description

- **[*n*] expression**

  An arbitrary expression. Each *expression* can be duplicated *n* times by specifying the repeat value within square brackets ([ ]) before the *expression* that is to be repeated. The repeat value, if present, must be a nonforward-referencing absolute value. If the repeat value and its surrounding square brackets are not used, then a single copy of the expression is emitted.

The **.short** and **.uashort** directives generate initialized half-words containing the 16-bit values represented in *expression*. Absolute values that will not fit within two bytes are truncated on the left. Only 16-bit relocatable or absolute values are allowed.

For the **.short** directive, if the assembly program counter is not on a two-byte boundary and if data alignment is turned on (as indicated by the **ALDATA** assembler symbol), the assembler program counter will be aligned to a two-byte boundary before the data is emitted. Any label on the same line as the directive will have its offset value adjusted to the aligned boundary.

The **.uashort** directive never checks for data alignment.

Examples

```
    .short [4]1
```

In the example above, four half-words of **1**s are generated.

.space — Reserves Bytes

Syntax

.space ***value***

Description

- **value**

  An expression that evaluates to an absolute value at assembly time. The expression cannot contain forward references.

The **.space** directive reserves storage in a section. Specifically, it is used to put space between elements in an object module. If the current section type is **bss**, no physical space is actually consumed in the output object module. If the **.space** directive is encountered in a **data** section, the intervening space is initialized with zeros. If the section is a **text** section, the space is initialized with instructions that branch to the next instruction (although some zeros may be emitted in order to align to a 4-byte boundary before emitting the branch instructions).

The assembly program counter will be increased by *value* for the current section. Thus, alignment of the next data directive or instruction with the **.align** directive should be considered because the program counter may be left at an undesirable boundary.

Example

```
array: .space 100
```

In this example, 100 bytes are allocated to the symbol name **array**.

.stitle — Sets Listing Subtitle

Syntax

.stitle "***string***"

Description

- **string**

  Any text.

The **.stitle** directive inserts the contents of *string* immediately below the header at the top of each assembler listing page that follows the directive. The header is determined by the **.title** directive. The subtitle string can also be accessed through the use of the **LSTITLE** assembler symbol.

The *string* specified will first appear on the listing page following the page that contained the **.stitle** directive until the end of the listing or until another **.stitle** directive is found.

If the listing command line option (**-l**) is not used, this directive does not have any obvious result because no assembler listing is produced.

Example

```
    .stitle "Title Below Header"
```

In this example, the string **Title Below Header** appears immediately below the header at the top of each following assembler listing page.

.string — Stores Strings

Syntax

.string "***string***" [, "*string*"]...

Description

- **string**

  Any text up to 8200 characters.

The **.string** directive initializes an area of storage to the value of *string*. A null character is appended to each *string* by the assembler. Alignment should be considered for any following directives. For example, if the length of *string* is 5 bytes and data alignment is turned off, any **.short**, **.long**, **.double**, or **.float** directives that follow will be misaligned. Likewise, if the next statement is an instruction, the assembler will also warn about misalignment since instructions must be aligned on a word boundary. The resulting output is unlikely to be valid under those conditions.

Mixing instructions and data in the same section is not prohibited. However, it is not recommended because of the affect of alignment on instructions.

Use the **.align** directive to properly align data directives.

Example

```
.string "odd-sized string"
.align 4
```

In this example, the ASCII character sequence **odd-sized string** is generated, followed by a NULL fill byte. Extra bytes are emitted by the **.align** directive to ensure word-alignment of subsequent statements.

.supervisoroff — Turns on Warning for Supervisor-Level Instructions

Syntax

.supervisoroff

Description

The **.supervisoroff** directive is used to signal the assembler that any instruction following this directive is to be executed in user mode. When supervisor mode is off, the execution of privileged instructions results in a run-time exception and a warning message for each supervisor-level instruction that is encountered. Supervisor mode will remain off until a **.supervisoron** directive is issued.

Off is the default supervisor setting for the assembler.

This directive has no effect if warning messages are suppressed through the use of the **-Q** or **-w** command-line option.

Example

```
mtmsr r3 #warning
.supervisoron
mtmsr r3 #no warning
.supervisoroff
mtmsr r3 #warning
```

The example demonstrates the effect of the supervisor mode on an instruction that may only be executed in supervisor mode.

.supervisoron — Turns off Warning for Supervisor-Level Instructions

Syntax

.supervisoron

Description

The **.supervisoron** directive is used to signal the assembler that any instruction following this directive is to be executed in supervisor mode. When supervisor mode is on, the execution of privileged instructions does not result in a run-time exception and warning messages are not issued for supervisor-level instructions until the end of the file or a **.supervisoroff** directive is issued. It is up to the user to control when the warnings are turned on again in a given module. Not issuing a **.supervisoroff** directive may result in some missed warnings from the assembler and potential run-time exceptions when that code is executed.

This directive has no effect if warning messages are suppressed through the use of the **-Q** or **-w** command-line option.

Example

```
mtmsr r3#warning
.supervisoron
mtmsr r3#No warning
.supervisoroff
mtmsr r3#warning again
```

The example demonstrates the effect of the supervisor mode on an instruction that may only be executed in supervisor mode.

.title — Sets Listing Title

Syntax

.title "***string***"

Description

- **string**

  Any text. The text appears as the title at the beginning of each page of the listing.

The **.title** directive inserts the contents of *string* at the top of each following assembler listing page. The default title defined by the assembler is **ASSEMBLY LISTING OF** *filename*. For a user-specified title to appear on the first page of the output listing, this directive must be the first statement in the program before all other lines, including comments. The title string can also be accessed through the use of the **LTITLE** assembler symbol.

If the listing command line option (**-l**) is not used, this directive does not have any obvious result because no assembler listing is produced.

Example

```
.title "Test Program"
```

In this example, the title **Test Program** appears at the top of each following assembler listing page.

.using — Associates Expression With Base Register

Syntax

.using ***expression*,*value***

Description

- **expression**

  An expression that is either a section name or an address within a section. The section name can be associated with either a relocatable section or a section created with the .dsect directive. The expression can contain forward references.

- **value**

  An expression that evaluates to an absolute value at assembly time. This value may be a general register symbol (such as r3) or an absolute value that is between 0 and 31, inclusively. The expression cannot contain any forward references.

The **.using** directive creates an association between a general register and an expression that represents an address or offset within a section. When an instruction is used that takes a displacement and a base register, and if the base register is not specified, then the assembler checks to see if there is an association created between a general register and an address/offset that is within 32KB of the displacement. If such an association can be found, then the matching general register will be used as the base register for the instruction. The displacement is also modified to be an offset from the expression that is assumed to be stored in the general register. If no base register can be found that has an associated expression that is within 32KB of the displacement, an error is generated.

If more than one base register can be found that is within 32KB of a displacement operand of an instruction, the lower numbered general register will be used.

The **.using** directive only creates an assembly time association between a general register and an expression. The general register must still be loaded with that value through other assembly instructions.

If the general register used with a **.using** directive is the same as one used in an earlier **.using** directive, the old expression association is forgotten and the new expression will be matched. If no **.using** directives exist for a given general register, it is presumed that no expression association exists for that general register. Associations between an expression and a general register can be cleared by using the **.drop** directive.

Example

```
.using dummy,r5
lhau r3,d1# Assuming d1 is a label in a dsect
         # section
.drop r5
lhau r3,d1# error, no matching .USING register
```

In this example, the general register **r5** is assumed to contain the address of some data structure described by the directives belonging to a **dsect** section. When a label from that section is used in an instruction with no base register indicated, the assembler will supply the missing base register **r5**. The assembler will also adjust the expression to be an offset from the value assumed to be in the base register. Note that once the association between the dsect and the general register is cleared through the **.drop** directive, the assembler will be unable to determine what base register should be used for the instruction and will emit an error.

.vbyte — Initializes One to Four Bytes

Syntax

.vbyte ***size***, ***expression***

Description

- **size**

  An expression that evaluates to an absolute value at assembly time. This value cannot contain any forward references. This value must be from 1 to 4.

- **expression**

  An absolute or relocatable expression.

The **.vbyte** directive initializes one to four bytes using the expression indicated in the directive. If the expression will not fit within the indicated number of bytes, it will be truncated on the left. The expression may only be relocatable if the indicated size is two or four bytes and if the relocation type fits within the indicated size.

No automatic data alignment occurs with the use of this directive.

Example

```
.vbyte 2,35
```

In this example, two bytes will be emitted that contain the value **35**. This example would be equivalent to using:

```
.short 35
```

.width — Sets the Maximum Line Width

Syntax

.width ***value***

Description

- **value**

  An expression that evaluates to an absolute value at assembly time. This value cannot contain any forward references and must be between 40 and 4096.

The **.width** directive sets the maximum page width in the listing. No line will be emitted that has a width larger than the indicated value. The line will either wrap to the next line or will be truncated at the maximum page width boundary, depending on the line wrap setting.

This directive is the same as using the **l** flag in the **.lflags** directive. This directive may only appear once in the source file and affects the entire listing output.

The **.width** directive sets the **LWIDTH** assembler symbol value to the indicated value.

The default line width is **<u>132</u>**.

If the listing command line option (**-l**) is not used, this directive is ignored because no assembler listing is produced.

Example

```
.width 80
```

In this example, the page width is set to **80**. Any lines emitted will have a line width that is less than or equal to this value.

.wrapoff — Turns Off Line Wrapping

Syntax

.wrapoff

Description

The **.wrapoff** directive turns off the automatic wrapping of long lines in the listing. If a line exceeds the maximum width of a listing line (as set by the **.width** directive), the line will be truncated in the listing at the maximum line width boundary.

The **.wrapoff** directive will set the **LWRAP** assembler symbol value to zero (0).

The default setting for the assembler is to perform wrapping of long lines.

If the listing command line option (**-l**) is not used, this directive is ignored because no assembler listing is produced.

Example

```
    .wrapoff
```

In this example, any lines emitted after this directive that exceed the maximum line width will be truncated in the listing.

.wrapon — Turns On Line Wrapping

Syntax

.wrapon

Description

The **.wrapon** directive turns on the automatic wrapping of long lines in the listing. If a line exceeds the maximum width of the listing line (as set by the **.width** directive), the line will stop at the maximum line width boundary and be continued on the next listing line. No line count indicator will be placed at the start of the extra line. For page length purposes, the extra line is tracked.

This is the default setting for the assembler.

The **.wrapon** directive will set the **LWRAP** assembler symbol value to one (1).

If the listing command line option (**-l**) is not used, this directive is ignored because no assembler listing is produced.

Example

```
    .wrapon
```

In this example, any lines emitted after this directive that exceed the maximum line width will be continued on the next line.

Chapter 7
Section Directives

A section is the smallest entity to which the linker can allocate memory, perform relocation, and assign absolute addresses. All code and data appear in sections in an Executable and Linking Format (ELF) object file. If you do not specify the type of section in which code and data are to appear, the assembler will default to a section called **.text**.

Table 7-1 describes a section's possible attributes.

Table 7-1. Section Attributes and Contents

Attribute	Content
A	The contents of the section will be written to the output file and will be used to initialize memory. If not set, memory will be set aside but not initialized. The absence of this attribute indicates a **bss** section.
W	The contents of the section will be writable. If not set, section contents are not considered to be modifiable. Any section with this attribute is normally placed in RAM. Any section without this attribute is normally placed in ROM.
X	The contents of the section will be executable. If not set, section contents cannot be executed. Any section with this attribute is normally expected to be placed in ROM.

The write and execute attributes affect how the section is processed by the linker. They will not prevent the user from putting instructions into writable sections or putting writable data into executable sections. However, putting items into sections that do not conform to the above conventions can result in problems at run time.

A section name is treated in the same way as any other symbol and can have up to 8192 characters.

All sections in an ELF object file are relocatable. They are assigned a physical address only by the linker. "Dummy" sections can be created and used within assembly files, but they are not written into the output file and are only useful as templates for data structures.

Section Directive Summary

The assembler section directives are shown in Table 7-2.

Table 7-2. Assembler Section Directives

Section Directive	Function
.comm	Declares a common symbol
.dsect	Declares a dummy section
.org	Sets program counter value
.pop_sect	Uses previously saved section.
.prev	Uses previous section
.push_sect	Saves section and declares a relocatable section.
.sect	Declares a relocatable section

Section Directives Descriptions

The assembler section directive descriptions are organized alphabetically on the following pages. Section directives fall into three categories as shown in Table 7-3.

Table 7-3. Assembler Directives by Category

Section Directive	Function
Declaring Sections	
.dsect	Declares a dummy section
.sect	Declares a relocatable section
Using Predefined Sections	
.comm	Declares a common symbol
Using Sections	
.org	Sets program counter within the current section
.pop_sect	Uses previously saved section.
.prev	Switches to the previously used section
.push_sect	Saves section and declares a relocatable section.

.comm — Declares a Common Symbol

Syntax

.comm ***symbol_name***,***size*** [,*alignment*]

Description

- **symbol_name**

  An identifier representing a symbol that has not already been defined in the current module, except possibly with the .extern or .globl directive.

- **size**

  An arbitrary expression that evaluates to an absolute value. The expression cannot contain any forward references. The *size* indicates the number of bytes of memory that must be reserved for *symbol_name*.

- ***alignment***

  An arbitrary expression that evaluates to an absolute value. The expression cannot contain any forward references. The *alignment* indicates the address alignment on which that *symbol_name* must be created. If this value is missing, the default alignment is used. This value is normally 4, but it can be changed with the **-f bn** flag.

The **.comm** directive declares uninitialized static variables and common blocks.

The identifier named in *symbol_name* must not be allocated space in the current module. If the identifier is made global and another module does not explicitly define the symbol, then the linker will allocate space for the symbol in the **.bss** section. The linker selects the largest *size* value that it finds in all of the modules that define that common symbol and allocates that many bytes.

This process makes it possible to declare a common block in many different modules and to initialize it optionally in at most one module. When a module defines a global symbol that occurs in a **.comm** directive in another module, the symbol retains its value and the common references refer to this symbol.

When used by itself, this directive creates the symbol in the local module scope only. When defined in this way, no other module can reference this symbol directly. If the symbol is to be accessed by other modules, this symbol must be used in a matching **.extern/.globl** directive.

Example

```
.comm tstcomm,100
.extern tstcomm
```

In this example, the symbol **tstcomm** becomes a common symbol. At link time, if **tstcomm** has not been declared in the **.extern/.globl** directive of another module as a normal symbol, the linker will allocate at least 100 bytes for it in the **.bss** section.

.dsect — Declares a Dummy Section

Syntax

.dsect ***section_name***

Description

- **section_name**

  An identifier that is the dummy section name. It must not be the same as any symbol or relocatable section within the assembly file. The name cannot begin with a period (.).

The **.dsect** directive defines a dummy section. It is like a structure template or definition that only indicates positioning of its members, but doesn't actually allocate any memory. This directive is present for convenience, but performs no functions that cannot be handled with the **.set** directive, albeit more awkwardly.

A dummy section's contents can be accessed from D-form load or store instructions. This is allowed when the dummy section or a label defined in a dummy section is associated with a base register through a **.using** directive. The **.dsect** directive is much like the **.sect** directive in that it leaves the previous section and alters the location counter. However, a dummy section cannot contain instructions. The **.align**, **.org**, and **.space** directives are allowed since they only affect the program counter and do not produce any output. Data directives are allowed, but only with absolute expressions. No data is actually emitted; only the program counter is adjusted.

The **.dsect** directive takes a section name as an argument. If a dummy section is opened a second time, it is set to the same location counter that it had at the end of its last use. The dummy section ends with a **.sect**, **.prev**, or another **.dsect** directive. Each named dummy section is uniquely defined.

Example

Dummy sections are typically used in combination with the **.using** directive. The **.using** directive is used to indicate that a general register has been loaded with the address of some section or symbol. If expressions are used in certain instructions that contain addresses that are related to the expression which is stored in a general register, the assembler will modify the instruction’s operands so the source or destination register is the indicated general register and the displacement is an offset from the expression in the general register. The following is an example of how it might be used:

```
.dsect mystruct
field1: .space 4 # A long value
field2: .space 2 # A short value
field3: .space 8 # A double value

.sect .text

# Tell assembler that r6 contains address of
# the start of a structure
.using mystruct, r6

# Set r6 to the address of a
# structure that exists somewhere
# in memory.
addis r6,r0,ha(astruct)
addi r6,r6,lo(astruct)

# Now, reference fields in that structure.
# This instruction will get the double value
# from the indicated structure.
# This instruction is the equivalent of
# the following: lfd f4,6(r6)
lfd f4,field3
```

.org — Sets Program Location Counter

Syntax

.org ***expression***

Description

- **expression**

  An arbitrary expression that evaluates to an absolute value, or a relocatable value that is relative to the current section. The expression may not contain any forward references.

The **.org** directive sets the program location counter to the indicated expression. If the expression is a relocatable value, the expression must be relative to the current section. The program location counter is set to the offset portion of the expression, relative to the start of the section in this module. Likewise, if the expression is an absolute value, the program counter is set to that value.

If the expression is a negative value or less than the current program counter, an error is emitted and the directive is ignored.

This directive is normally used to set aside memory or to set the location counter to a specific offset within the section. The first use is similar to the **.space** directive, while the second use is similar to the **.align** directive.

Example

```
.sect .text
nop
.org $+20
nop
```

The **.org** directive adds 20 to the current program location counter. The **nop** instruction will then be emitted at the new section offset.

.pop_sect — Uses Previously Saved Section

Syntax

.pop_sect

Description

The **.pop_sect** directive assembles statements following it into the relocatable or dummy section saved at an earlier point through the use of the **.push_sect** directive. If used before any sections are saved, the current section continues to be used. The order of sections being restored is the reverse of the order that they are saved. There is no limitation on the number of sections that can be saved. Also, there can be any number of sections defined between the saving of a section (through the **.push_sect** directive) and the restoring of that section (through the **.pop_sect** directive).

Example

```
.sect .text
NOP
.push_sect .data
.long 1
.pop_sect
```

In this example, any statements following the **.pop_sect** directive will be assembled into the **.text** section.

.prev — Uses Previous Section

Syntax

.prev

Description

The **.prev** directive assembles statements following it into the relocatable or dummy section defined before the current section. If used before any sections are defined, the default **.text** section remains the current section.

Example

```
 .sect .text
NOP
 .sect .data
 .long 1
 .prev
```

In this example, any statements following the **.prev** directive will be assembled into the **.text** section.

.push_sect, .sect — Declares a New Section

Syntax

.push_sect ***section_name***[*section_type*][,*alignment*]

.sect ***section_name***[*section_type*][,*alignment*]

Description

- **section_name**

  An identifier that is the name of a section. If the first character is a period (.), it must be the name of one of the predefined sections. When creating a new section that does not begin with a period, it must not be the name of an existing .dsect section or a symbol. Section names are case-sensitive.

- ***section_type***

  One or more attributes that indicate the section's contents and whether the section is to be allocated. Valid attributes are A (allocate), W (write), and X (executable). Square brackets ([ ]) are required when specifying attributes.

- ***alignment***

  An arbitrary expression that evaluates to an absolute value. The expression may not contain any forward references. The *alignment* indicates the address alignment that the *section_name* must be created on. If this information is missing, the default alignment is four for executable sections and one for all other sections.

The **.push_sect** and **.sect** directives define a relocatable section that can be used to contain data or instructions. When initially defined, a section can be given a *section_type* and *alignment*. Any following definition of the section will retain or modify the initial values of the *section_type*, *alignment*, and location counter.

The only difference between the **.push_sect** and **.sect** directives is that the **.push_sect** directive pushes information about the previously defined section onto a stack for later use. At some later time, a **.pop_sect** directive may be used to return to that saved section. The order of sections being restored is the reverse of the order in which they were saved. There is no limitation on the number of sections that can be saved. Also, there can be any number of other sections defined between saving a section with the **.push_sect** directive and restoring it with the **.pop_sect** directive.

The *section_type* associated with a section is used to indicate what the contents of a section would be. There are three different attributes that a section may have:

- The **A** attribute indicates that the section is to be allocated space in the output file. If this flag is missing, the section is treated like a **bss** section. Only the data, **.align**, **.org**, and **.space** directives can be used within a **bss** section. No instructions are allowed. If this flag is present, the section's contents will be initialized in memory to the values contained within the section. If **A** is not set, the **W** attribute is required.

- The **W** attribute indicates that the section's contents are writable. This type is normally used to indicate that the section contains modifiable data. Normally, this section would be placed in RAM. If this attribute is missing, the section would normally be placed in ROM.
- The **X** attribute indicates that the section contains instructions. This normally means that some, but not necessarily all, of the section's contents are instructions. Some read-only data may also exist within the same section. This section would normally be placed in ROM. The **A** attribute must always be used when this attribute is present.

If no *section_type* is indicated, then the section is presumed to be allocated in memory and to contain instructions ([**AX**]). Note that reopening a section with a different section type than any previous definition will result in the new attributes being added to any previous attributes. Thus, it is possible to open a section as being only allocated, but later modifying the section's attributes to be executable as well. Any attributes from earlier definitions cannot be removed, only added to. The only case where a section's attributes may not be modified is when a section is created as a **bss** section. Once defined as type **bss**, the section must remain a **bss** section.

The assembler predefines many sections. The names of these sections all begin with a period (**.**). No other sections can be created with names that begin with a period. These predefined sections have a default *section_type* and are aligned depending upon its expected contents. If these sections are used with the **.sect** directive, the *section_type*, if specified, must match the indicated section type exactly, except for the definition of the **.sdata2** section. The **.sdata2** section is initially only allocated, but may be defined to be both allocated and writable.

Table 7-4. Predefined Section Attributes and Alignment

Section Name	Section Type	Alignment
.bss	W	1
.data	AW	1
.data1	AW	1
.fini	AX	4
.init	AX	4
.PPC.EMB.sbss0	W	1
.PPC.EMB.sdata0	AW	1
.rodata	A	1
.rodata1	A	1
.sbss	W	1
.sbss2	W	1
.sdata	AW	1
.sdata2	A or AW	1

Table 7-4. Predefined Section Attributes and Alignment (cont.)

Section Name	Section Type	Alignment
.text	AX	4

Note that the **.init** and **.fini** sections are only for use by the Microtec CCCPPC C++ Compiler.

Because these section names are predefined, a statement like the following would generate a warning:

```
    .sect .bss[AX]
```

Example

```
    .sect kernel[AX],16
```

In this example, a relocatable section named **kernel** is declared. It is allocated in memory and contains instructions. The linker will place this section on at least a 16-byte boundary in memory.

Chapter 8
Macros

Macro Functions

Macro functions can be altered by changing the source code in only one location: the macro definition. A macro definition consists of three parts: a heading, a body, and a terminator. This definition must precede any macro call.

A macro can be redefined at any place in the program; the most recent definition is used when the macro is called. A standard mnemonic (such as **ORI**) can also be redefined by defining a macro with the same name. In this case, all subsequent uses of that instruction in the program cause the macro to be expanded.

Macro Heading

The heading, which consists of the directive **.macro**, gives the macro a name and defines a set of formal parameters. For more information on the macro heading, see the **.macro** directive in this chapter.

Macro Body

The first line of code following the **.macro** directive is the start of the macro body. The macro statements that make up the macro body are placed in a macro file for use when the macro is called. All assembly statements except **.end** are also valid within the macro body.

No statement in a macro definition is assembled at definition time. Statements in the macro definition are stored in the macro file until called, at which time they are inserted in the source code at the position of the macro call.

The name of a formal parameter specified on the **.macro** directive can appear within the macro body in any field. When the macro is called, all parameter names appearing in the macro body will be substituted by the actual parameter values from the macro call, except in a comment statement or in the comment field of a statement.

A formal parameter in the macro body is indicated by using the name of the parameter. To use the parameter name without replacement, prefix the name with a backslash (\). The backslash will be removed from the output.

Use angle brackets (<>) to separate text that is to be concatenated in the output after parameter replacement. This is important if a formal parameter occurs immediately after text that does not change. The angle brackets are always removed from the output.

The **!MLIST** symbol can be used in a macro definition and is replaced by a value that indicates the actual number of parameters used with the macro call (empty or missing parameters are not counted).

The assembler symbols **BCOUNT** and **MCOUNT** can be used to determine how many macro calls or repeat blocks have occurred within the assembly source. Both symbols are initially set to 0. Whenever a macro call occurs or a repeat block is duplicated, the **MCOUNT** symbol is incremented. The **BCOUNT** symbol is assigned the value of the **MCOUNT** symbol at that time. The **BCOUNT** symbol value is restored after nested macro calls or repeat blocks while the **MCOUNT** symbol is always the maximum number of macro calls or repeat blocks that have occurred so far. The values of these symbols may be accessed by prefixing the symbol with an exclamation mark (**!**).

Unique labels can be created in macros or repeat blocks by placing an exclamation mark immediately after each label. This character will signal to the assembler to replace the exclamation mark with an underscore (_) followed by the block count of the current macro or repeat block. The block count can be accessed through the assembler symbol **BCOUNT**. This form of label definition or reference cannot be used outside of macros or repeat blocks.

Macro Terminator

The **.mend** directive terminates the macro definition.

Nested Macro Definition

A nested macro definition is contained completely within the body of another macro definition. The nested macro does not become defined until the outer macro is called. A nested macro can even be used to redefine itself. The assembler knows which **.mend** or **.endrept** directive to use to terminate the macro by maintaining a count of **.macro** or repeat directives encountered. Therefore, you must ensure that each nested definition is properly terminated within the outer macro body.

Example 8-1. Nested Macros

```
   .macrofoo,par1,par2# two parameters
lab1!:.longpar1   # par1 will be substituted
   .long dummypar1# no substitution
lab2!:.longdummy<>par2# par2 will be substituted
   .long !MLIST   # !MLIST is replaced by a value
                  # indicating the actual number
                  # of parameters in the macro
                  # call.
   .mend
```

In this example, the macro body consists of the first line of code following the **.macro** directive to the line of code preceding the **.mend** directive.

Calling a Macro

Syntax

[*label*:] ***name*** [*parameter* [,*parameter*]...]

Description

- ***label***

  Assigns the current value of the current program counter to label.

- **name**

  Names the macro to be called. This name must be defined by the .macro directive or an error message is generated.

- ***parameter***

  Specifies a parameter to be passed to the macro. A parameter can be a constant, symbol, expression, character string, or any other text separated by commas. The maximum number of parameters supported in a macro call is 35.

A macro can be called by encoding the macro name in the operation field of the statement. Since macro names are case-sensitive, the exact name used in the definition must be used to call the macro. The parameters in the macro call are actual parameters, and their values may be different from the formal parameters used in the macro definition. The actual parameters are substituted for the formal parameters in the order in which they are written. Commas (**,**) can be used to skip a parameter position. In this case, the parameter will be null (that is, containing no actual characters). The formal parameter corresponding to a null actual parameter is removed during macro expansion. Any parameter not specified will be null. The parameter list is terminated by a newline or a pound sign (#).

All actual parameters are passed as character strings into the macro definition statements. Thus, symbols are passed by name and not by value. In other words, if a symbol's value is changed in a macro expansion, it will also have the new value after the expansion. Any **.set** directives within a macro body may alter the value of parameters passed to the macro.

Since the comma (**,**) is used as a separator for arguments, it may only be used as part of an argument outside of a string by putting a backslash before the comma character. The comma will become part of the argument and the backslash (\) will be removed. It should be noted that any backslash character outside of a string in an argument will be removed while the character following the backslash (which might be another backslash) will be retained as part of the argument.

Macro parameter substitution is determined by the following rules:

1. Parameter names that appear as tokens (symbols) will be substituted. For example, **par1** and **par2** in **if par1<par2** will be substituted.

2. Comments will be ignored. No substitution takes place.

3. Parameter names may be concatenated to other names or embedded within other names by using angle brackets (<>) as a concatenation indicator.

4. All occurrences of a single backslash (\) before a symbol will disappear. The backslash will simply be treated as a concatenation character whether or not a parameter appears in front or in back of it. If a parameter name follows a backslash, it will not be replaced. If in front of a backslash, the parameter will be expanded unless it is also prefixed with a backslash.

5. A pair of backslashes are treated as one. The backslash serves as a break character for the purpose of substitution. Only one pass is allowed, so a series of backslashes will not be reduced. In other words, an occurrence of double backslashes in a string or as a part of a token becomes a single backslash when the token is returned.

6. Within quoted strings, no substitution will take place. Rules 4 and 5 still apply.

Example 8-2. Sample Macro Call

```
.macrowords,par1,par2# Start of definition
.long par1
.long par2     # End of definition
.mend

words 3 , ABC >>2 # Right shift ABC twice
            # Macro call
```

In this example, the macro call is:

```
words 3 , ABC >> 2 # Right shift ABC twice
```

The first parameter is replaced by:

```
3
```

The second parameter is replaced by:

```
ABC>>2
```

The macro expands to:

```
.long3
.longABC>>2
```

A more extensive example of a macro definition, call, and expansion is shown in Figure 8-1.

Figure 8-1. Macro Definition, Call, and Expansion

```
Microtec ASMPPC V1.1 Page 1 Thu Sep 21 18:35:08 1995
ASSEMBLY LISTING OF ch8.71.s

Command line: asmppc -l ch8.71.s
Line   Address  Opcode         Source
1                                     #
2                                 # The following macro loads a register with constant
3                                 # value. The macro name is load_reg. It has two parameters.
4
5                                 # Macro definition
6                                     .macro load_reg,reg,val
7                                     .if val >= -0x80 && val <= 0x7f
8                                     addi reg,0,val      # value will fit within 16 bits
9                                     .mexit # We won't have to see the      rest of the macro
10                           .else
11                           addis reg,0,ha(val) # get upper 16-bits of value
12                                               # This value is adjusted if adding
13                                               # the sign-extended low 16-bits to
14                                               # this will result in overflow
15                           addi reg,reg,lo(val)# merge in the lower 16-bits
16                                               # This will be sign-extended, so
17                                              # the result will be the correct value.
18                           .mend
19
20                           .lflags mc          # List macro and if-statement expansions
21
22                           load_reg r5, 53     # Should fit with no problem in 2 bytes.
22.1                         .if 53 >= -0x80 && 53 <= 0x7f
22.2   00000000 38 A0 00 35  addi r5,0,53        # value will fit within 16 bits
23                           load_reg r8, -2     # Should fit with no problem in 2 bytes.
23.1                         .if -2 >= -0x80 && -2 <= 0x7f
23.2   00000004 39 00 FF FE  addi r8,0,-2        # value will fit within 16 bits
24                           load_reg r9, 80000  # Will require a full 32-bits
24.1                         .if 80000 >= -0x80 && 80000 <= 0x7f
24.2                         addi r9,0,80000     # value will fit within 16 bits
24.3                         .mexit              # We won't have to see the rest of the macro
24.4                         .else
24.5   00000008 3D 20 00 01  addis r9,0,ha(80000)# get upper 16-bits of value
24.6                                             # This value is adjusted if adding
24.7                                             # the sign-extended low 16-bits to
24.8                                             # this will result in overflow
24.9   0000000C 39 29 38 80  addi r9,r9,lo(80000)# merge in the lower 16-bits
24.10                                            # This will be sign-extended, so
24.11                                            # the result will be the correct value.
25                            .end
```

Special Characters

During the macro expansion, the assembler recognizes certain characters as having special meaning. The angle bracket (<>) characters are used to concatenate the text on a definition line with any actual parameter. During macro expansion, angle brackets (<>) immediately preceding or immediately following a formal parameter are removed, and the substitution of the actual parameter occurs at that point. The angle bracket (<>) characters are removed, regardless of location, unless they occur within strings.

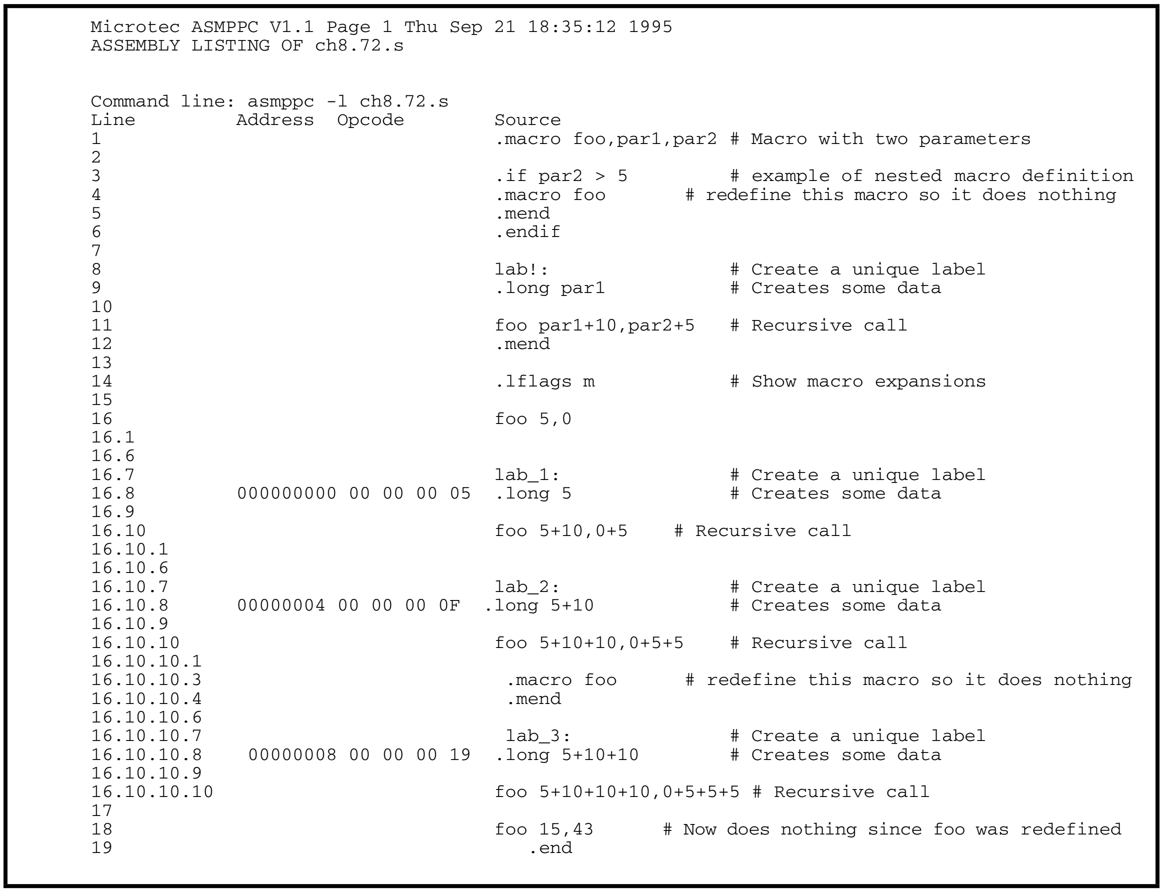
Sample Listing

The following is a description of the macro named **foo** listed in Figure 8-2:

- The outer macro definition consists of lines 1 through 12.
- The macro is named **foo** and has two parameters, **par1** and **par2**.
- Line 9 shows how a parameter is substituted.

- Lines 3 through 6 represent code that gets assembled on the condition that the value represented by **par2**, after being substituted and evaluated, is greater than 5.
- Lines 4 and 5 redefine **foo** to be an empty macro.
- Line 11 is a recursive macro call.
- Line 14 tells the assembler to list macro expansions.
- Line 16 is the actual macro call, and lines **16.1** through **16.10.10.10** show the actual lines that are generated as a result of the macro call.
- Lines **16.7**, **16.10.7**, and **16.10.10.7** show local labels created during each macro call. The value appended to the label is created using the **BCOUNT** assembler symbol's value during that macro call.
- The third recursive call to **foo**, on line **16.10.10.10**, does nothing since the **foo** macro was redefined on lines **16.10.10.3** and **16.10.10.4**.
- Line 18 is another call to the **foo** macro. Since foo was redefined to be empty during the previous macro call, this macro call does nothing.

Figure 8-2. Sample Listing for a Macro



```
Microtec ASMPPC V1.1 Page 1 Thu Sep 21 18:35:12 1995
ASSEMBLY LISTING OF ch8.72.s


Command line: asmppc -l ch8.72.s
Line         Address  Opcode        Source
1                                   .macro foo,par1,par2 # Macro with two parameters
2
3                                   .if par2 > 5         # example of nested macro definition
4                                   .macro foo       # redefine this macro so it does nothing
5                                   .mend
6                                   .endif
7
8                                   lab!:                # Create a unique label
9                                   .long par1           # Creates some data
10
11                                  foo par1+10,par2+5   # Recursive call
12                                  .mend
13
14                                  .lflags m            # Show macro expansions
15
16                                  foo 5,0
16.1
16.6
16.7                                lab_1:               # Create a unique label
16.8         000000000 00 00 00 05  .long 5              # Creates some data
16.9
16.10                               foo 5+10,0+5    # Recursive call
16.10.1
16.10.6
16.10.7                             lab_2:               # Create a unique label
16.10.8      00000004 00 00 00 0F  .long 5+10            # Creates some data
16.10.9
16.10.10                            foo 5+10+10,0+5+5    # Recursive call
16.10.10.1
16.10.10.3                           .macro foo      # redefine this macro so it does nothing
16.10.10.4                           .mend
16.10.10.6
16.10.10.7                           lab_3:              # Create a unique label
16.10.10.8    00000008 00 00 00 19  .long 5+10+10        # Creates some data
16.10.10.9
16.10.10.10                         foo 5+10+10+10,0+5+5+5 # Recursive call
17
18                                  foo 15,43      # Now does nothing since foo was redefined
19                                     .end
```

Macro Directives

The assembler macro directives listed in Table 8-1 are described in the following pages.

Table 8-1. Alphabetical Listing of the Macro Directives

Macro Directive	Function
.macro	Begins a macro definition.
.mdepth	Sets maximum macro recursion depth.
.mend	Ends macro definition.
.mexit	Terminates macro expansion.
.mexpoff	Turns off macro expansion in listing.
.mexpon	Turns on macro expansion in listing.
.mundef	Purges listed macros.

.macro — Begins a Macro Definition

Syntax

.macro ***macro_name*** [,*parameter*[,*parameter*]...]

Description

- **macro_name**

  An identifier that is the macro name.

- ***parameter***

  An identifier that is a formal parameter for the macro being defined.

The **.macro** statement or heading declares the macro name and its formal parameters. These parameters are bound to actual values when the macro is called, and text substitution of these parameters occurs in the macro body.

If *macro_name* is identical to a machine instruction or an assembler directive, the mnemonic's function is redefined by this macro definition. Once a mnemonic has been redefined, it can be returned to its standard function by the **.mundef** directive. If a macro and an instruction or directive have the same name, the instruction or directive can be accessed by prefixing the name with an exclamation mark (**!**). Since macro names are case-sensitive and instruction mnemonics and directives are not, an instruction or directive can be accessed by using a different case than is used in the macro definition.

A macro definition can redefine a previous definition. All macro calls from that point onward will use the new definition. It is not necessary to delete the previous definition with the **.mundef** directive before the new definition is defined.

The operand field of the **.macro** statement can contain the names of formal parameters in the order in which they will occur on the macro call. Each parameter is an identifier. Multiple parameters must be separated by commas. The identifiers used as formal parameters are known only to the macro definition and can be used as regular symbols outside the macro. These names will override any predefined symbols within the macro definition, such as register names (for example, **r1**). Each parameter name must be different from other parameter names for that macro.

Example

```
.macro load_reg,reg,val
```

In this example, the macro name is **load_reg** and its formal parameters are **reg** and **val**.

.mdepth — Sets Maximum Macro Recursion Depth

Syntax

.mdepth ***expression***

Description

- **expression**

  An arithmetic expression that evaluates to a nonnegative absolute value at assembly time. The expression may not contain any forward references.

The **.mdepth** directive sets the maximum number of macro calls or repeat blocks that can be nested within each other. If that limit is reached during macro expansions, an error will be emitted and the expansion will continue with the next line in the current macro or repeat block.

The **.mdepth** directive cannot be used within a macro or repeat expansion. The directive can be used multiple times within a source file. The **.mdepth** directive only affects macro expansions following that directive.

The **MAXDEPTH** assembly symbol is set to the value used with this directive.

The default maximum macro expansion depth is 100 (UNIX) and 25 (Windows).

Example

```
.mdepth 5
.macro store
   ...
store
   ...
.mend
store
```

In this example, the **.mdepth** directive terminates the macro expansion of the macro named **store** after the macro has called itself four times (making 5 the total number of nested macro calls). An error is generated indicating the maximum depth has been reached and expansion of the current macro will continue after the macro call.

.mend — Ends Macro Definition

Syntax

.mend

Description

The **.mend** directive terminates the macro definition. If a **.end** directive or end of file is found during a macro definition, the directive will terminate the macro definition and the assembly.

Example

```
.macro store,number
   ...
.mend
```

In this example, the **.mend** directive terminates the macro definition of the macro named **store**.

.mexit — Terminates Macro Expansion

Syntax

.mexit

Description

The **.mexit** directive provides an alternative method for terminating a macro expansion when a macro is called. During a macro expansion, a **.mexit** directive causes expansion of the current macro to stop and all code between the **.mexit** and the **.mend** for this macro to be ignored. If macro calls are nested, **.mexit** causes code generation to return to the previous level of macro expansion. Note that **.mexit** or **.mend** can be used to terminate a macro expansion, but only **.mend** can be used to terminate a macro definition.

Example

```
.macro store,number
   ...
.if number<10
.mexit
.endif
  ...
.mend
```

In this example, the code following the **.mexit** will not be assembled if the condition, **number<10**, is true.

.mexpoff — Turns off Macro Expansion in Listing

Syntax

.mexpoff

Description

The **.mexpoff** directive turns off the expansion of macro calls in the listing. If this directive is in force, the listing will only show a macro call, but will not show how the macro’s contents are expanded and processed. This directive can be useful to reduce the size of a listing when either large macros are used or recursion occurs within macro calls.

The **.mexpoff** directive is equivalent to using the **nm** option with the **.lflags** directive.

The **MEXP** assembler symbol is set to zero (0) if this directive is used.

The default for the assembler is to not expand macro calls in the listing.

If the listing command line option (**-l**) is not used, this directive is ignored because no assembler listing is produced.

Example

```
.mexpoff
.macro store,number
   ...
.mend
store 5
```

In this example, the **.mexpoff** directive will turn off macro expansion within the listing. When the macro named **store** is called, the listing will only show object code emitted by that expansion. There will be no indication in the listing as to how the parameter (5 in this example) was used in the macro.

.mexpon — Turns on Macro Expansion in Listing

Syntax

.mexpon

Description

The **.mexpon** directive turns on the expansion of macro calls in the listing. If this directive is in force, the listing will show a macro call, followed by lines containing each statement within the macro’s body. The macro’s contents are expanded and processed. This directive can be useful in showing how parameters are used in the expansion of a macro.

The **.mexpon** directive is equivalent to using the **m** option with the **.lflags** directive.

The **MEXP** assembler symbol is set to one (1) if this directive is used.

The default for the assembler is to not expand macro calls in the listing.

If the listing command line option (**-l**) is not used, this directive is ignored because no assembler listing is produced.

Example

```
.mexpon
.macro store,number
   ...
.mend
store 5
```

In this example, the **.mexpon** directive will turn on macro expansion within the listing. When the macro named **store** is called, the listing will show each line within the macro body and how any parameters within the body were updated. The statements within the body of the listing will be indicated by having a line number that is the same line number as the macro call, followed by a period, followed by the line number of the statement within the macro body.

.mundef — Purges Listed Macros

Syntax

.mundef ***macro_name***[,*macro_name*]...

Description

- **macro_name**

  An identifier representing the name of a defined macro.

The **.mundef** directive deletes the indicated macro definitions.

An error is generated if the indicated macro does not exist.

Defining a macro call with the same name as an earlier macro definition does the equivalent of a **.mundef** on the earlier macro definition. The new definition will be used from that point onward in the source file.

Example

```
.macro add,reg1,reg2,reg3
SUB reg1,reg2,reg3
.mend
add r1,r2,r3# a macro call that generates
            # a subtract instruction
Add r1,r2,r3# generates an add instruction
!add r1,r2,r3           # generates an add instruction
.mundef add # purge of macro definition
add r1,r2,r3# generates an add instruction
```

In this example, a macro named **add** is defined between the **.macro** and **.mend** directives as **SUB reg1,reg2,reg3**. The macro definition is then followed by a macro call that generates a subtract instruction.

Because macros are case-sensitive and instructions are case-insensitive, **Add r1,r2,r3** generates an add instruction. The macro **add** still exists, but **Add** is recognized as an add instruction.

The defined macro **add** is purged by **.mundef add**. Since the macro **add** no longer exists, the following **add** is then taken to be an add instruction.

Note that if a macro and an instruction have the same name, the macro name has precedence when used with the same name as the macro definition. If a name is prefixed with an exclamation mark (**!**), there will be no check for a macro with that name.

Predefined Macros

The Microtec PPC assembler provides a set of predefined LOAD/STORE macros for loading and storing values with the single macro call. Internally these macros use two instructions for the whole operation. These macros are listed in Table 8-2.

Table 8-2. Predefined LOAD/STORE Macros

Macro	Description
LD_ADDR	Load Address
LD_BYTE	Load Byte
LD_DOUBLE	Load Double Precision
LD_HWORD	Load Half Word
LD_IMM32	Load Immediate
LD_SINGLE	Load Single Precision
LD_WORD	Load Word
ST_BYTE	Store Byte
ST_DOUBLE	Store Double Precision
ST_HWORD	Store Half Word
ST_SINGLE	Store Single Precision
ST_WORD	Store Word

The Microtec PPC assembler also provides a set of predefined macros intended for moving local values within a C or C++ function. Internally these macros usually use one instruction for the operation, but could use none or two. These macros are listed in Table 8-3.

Note

The complete definitions of these macros can be viewed in the assembler listing by using the assembler directive **.lflags cm**.

Table 8-3. Predefined MOVE_LOCAL Macros

Macro	Description
MOVE_LOCAL	Move a 32-bit integer local value
MOVE_LOCAL_FD	Move a 64-bit double float local value
MOVE_LOCAL_FS	Move a 32-bit single float local value
MOVE_LOCAL_SB	Move a signed 8-bit integer local value

Table 8-3. Predefined MOVE_LOCAL Macros (cont.)

Macro	Description
MOVE_LOCAL_SH	Move a signed 16-bit integer local value
MOVE_LOCAL_UB	Move an unsigned 8-bit integer local value
MOVE_LOCAL_UH	Move an unsigned 16-bit integer local value

Finally, the Microtec PPC assembler provides a set of macros for loading a signed 8-bit value into a register. The PowerPC does not have a single instruction for doing this, so two instructions are needed. These macros are listed in Table 8-4.

Table 8-4. Predefined MOVE_SIGNED_CHAR Macros

Macro	Description
MOVE_SIGNED_CHAR	Load a byte value (offset) and sign extend
MOVE_SIGNED_CHAR2	Load a byte value (indexed) and sign extend
MOVE_SIGNED_CHAR_UPDATE	Load a byte value (offset) and sign extend
MOVE_SIGNED_CHAR_UPDATE2	Load a byte value (indexed) and sign extend

LD_ADDR — Load Address

Syntax

LD_ADDR ***reg***, ***variable_name***

Description

- **reg**

  A general-purpose register.

- **variable_name**

  A variable whose address is to be loaded.

This macro loads the address of a variable into a general-purpose register.

The following macro call:

```
LD_ADDR r4, variable
```

is replaced with the following instructions:

```
addis r4, 0, ha(variable)
addic r4, r4, lo(variable)
```

LD_BYTE — Load Byte

Syntax

LD_BYTE ***reg, variable_name***

Description

- **reg**

  A general-purpose register.

- **variable_name**

  A variable to be loaded.

This macro loads a single byte of data from memory into a register.

The following macro call:

```
LD_BYTE r4, byte_var
```

is replaced with the following instructions:

```
addis r4, 0, ha(byte_var)
lbz   r4, lo(byte_var)(r4)
```

LD_DOUBLE — Load Double-Precision

Syntax

LD_DOUBLE ***freg***, ***add_reg***, ***variable_name***

Description

- **freg**

  A floating-point register

- **add_reg**

  A general-purpose register used to construct the address of the variable.

- **variable_name**

  A double variable to be loaded.

The macro loads an 8-byte double-precision value from a memory location into a floating point register.

The following macro call:

```
LD_DOUBLE f0, r4, double_var
```

is replaced with the following instructions:

```
addis  r4, 0, ha(double_var)
lfd    f0, lo(double_var)(r4)
```

LD_HWORD — Load Half Word

Syntax

LD_HWORD ***reg***, ***variable_name***

Description

- **reg**

  A general purpose register.

- **variable_name**

  A variable to be loaded.

This macro loads two bytes of data from a memory location.

The following macro call:

```
LD_HWORD r4, short_var
```

is replaced with the following instructions:

```
addis r4, 0, ha(short_var)
lhz   r4, lo(short_var)(r4)
```

LD_IMM32 — Load Immediate

Syntax

LD_IMM32 ***reg***, ***IMM32***

Description

- **reg**

  A general purpose register.

- **IMM32**

  A 32-bit immediate value to be loaded.

This macro loads a 32-bit immediate value into a register.

The following macro call:

```
LD_IMM32 r4, 0x12345678
```

will be replaced by the following instructions:

```
addis r4, 0, hi(0x12345678)
ori   r4, r4, lo(0x12345678)
```

LD_SINGLE — Load Single-Precision

Syntax

LD_SINGLE ***freg***, ***add_reg***, ***variable_name***

Description

- **freg**

  Floating point register.

- **add_reg**

  A general purpose register used to construct the address of the variable.

- **variable_name**

  A single variable to be loaded.

This macro loads a 4-byte single-precision value from a memory location.

The following macro call:

```
LD_SINGLE f0, r4, float_var
```

is replaced with the following instructions:

```
addis  r4, 0, ha(float_var)
lfs    f0, lo(float_var)(r4)
```

LD_WORD — Load Word

Syntax

LD_WORD ***reg***, ***variable_name***

Description

- **reg**

  A general-purpose register.

- **variable_name**

  A variable to be loaded.

This macro loads 4 bytes of data from memory location into a register.

The following macro call:

```
LD_WORD r4, word_var
```

is replaced with the following instructions:

```
addis r4, 0, ha(word_var)
lwz   r4, lo(word_var)(r4)
```

MOVE_LOCAL - Move a 32-bit integer local value

Syntax

MOVE_LOCAL ***source,dest***

Description

- **source,dest**

  A general-purpose register or a stack reference. If the macro detects a ‘(‘character in the argument, it assumes it is a stack reference. Only one argument can be a stack reference. Otherwise, an illegal instruction is generated.

This macro generates an instruction to move a 32-bit value from source to dest, either using a load or store instruction or by using an ori (register move) instruction. If both arguments are registers and are identical, then no instruction is generated.

This macro is intended to be used with the C/C++ asm() statement to access local variables. Depending on the compiler options, an integer variable can either be in a register or on the stack. This macro provides a mechanism for accessing the variable regardless of compiler options.

The actual macro definition is as follows:

```
.macro MOVE_LOCAL,src,dest
 .ifins "(",'src'
 lwz dest,src
 .else
 .ifins "(",'dest'
 stw src,dest
 .else
 .ifnes 'src','dest'
 ori dest,src,0
 .endif
 .endif
 .endif
 .mend
```

Example

```
MOVE_LOCAL 24(r1),r4
```

In this example, the instruction lwz r4,24(r1) is generated.

```
MOVE_LOCAL r3,r31
```

In this example, the instruction ori r31,r3,0 is generated.

MOVE_LOCAL Variations

In additional to MOVE_LOCAL, six additional predefined macros are available to handle other sizes and types.

Four of the macros are available to handle byte and half-word moves. These macros are identical to the above MOVE_LOCAL macro, except that the load and store instructions (lwz and stw) are replaced with lbz or stb for byte, and lha or lhz or sth for halfword. Note that the PowerPC instruction set does not have a signed-byte load instruction, so an unsigned load byte instruction (lbz) followed by an extend sign byte instruction (extsb) is used to do a signed byte load. Two of the macros are used to handle floating point values in floating point registers; one for single precision and one for double precision. In addition to using different load/store instructions, the floating register move instruction (fmr) is used instead of ori. For single precision, lfs/stfs load/store instructions are used. For double precision, lfd/stfd instructions are used.

Table 8-5. MOVE_LOCAL Macro Instructions

Macro Name	Load Instruction	Store Instruction	Move Instruction
MOVE_LOCAL	lwz	stw	ori
MOVE_LOCAL_SB[1]	lbz/extsb	stb	ori
MOVE_LOCAL_UB	lbz	stb	ori
MOVE_LOCAL_SH	lha	sth	ori
MOVE_LOCAL_UH	lhz	sth	ori
MOVE_LOCAL_FS	lfs	stfs	fmr
MOVE_LOCAL_FD	lfd	stfd	fmr

[1] Since the PowerPC architecture does not have a signed byte load instruction, an unsigned byte load (lbz) is used and extsb generates the sign extension

MOVE_SIGNED_CHAR - Move a signed 8-bit integer into a register

Syntax

MOVE_SIGNED_CHAR dest,src

MOVE_SIGNED_CHAR2 dest,src1,src2

MOVE_SIGNED_CHAR_UPDATE dest,src

MOVE_SIGNED_CHAR_UPDATE2 dest,src1,src2

Description

- **src**

  The src is an expression that represents the Register Indirect with Immediate Index addressing mode. This should be a signed 16-bit immediate value followed by a general-purpose register within parentheses.

- **src1,src2**

  The two sources indicate general-purpose registers, whose contents are added together to form an address. This is referred to as the Register Indirect with Index addressing mode.

- **dest**

  A general-purpose register that is to hold the sign-extended 8-bit value.

These macros move an 8-bit value into a general-purpose register and then sign-extend that value. These macros are necessary because the PowerPC instruction set does not provide a single instruction that performs this action.

The differences between these macros is that two of the definitions use one form of addressing mode while the other two definitions use another. The choice of a macro will depend upon how the memory address of the 8-bit value is to be indicated.

Another difference between these macros is that two of the macros will modify one of the source general-purpose registers with the address of the 8-bit value that was loaded. The other two instructions do not modify the source register or registers.

Table 8-6. MOVE_SIGNED_CHAR Macro Instructions

Macro Name	Load Instruction	Sign Extend Instruction
MOVE_SIGNED_CHAR	lbz	extsb
MOVE_SIGNED_CHAR2	lbzx	extsb
MOVE_SIGNED_CHAR_UPDATE	lbzu	extsb
MOVE_SIGNED_CHAR_UPDATE2	lbzux	extsb

Example

```
    MOVE_SIGNED_CHAR r4,35(r2)
```

In this example, the macro used the ‘lbz’ instruction to load an 8-bit value into r4 from the memory location represented by adding 35 to the contents of r2. The 8-bit value in r4 is then sign-extended.

ST_BYTE — Store Byte

Syntax

ST_BYTE ***reg***, ***addr_reg***, ***variable_name***

Description

- **reg**

  A general-purpose register containing the byte to be stored.

- **addr_reg**

  A general-purpose register used to construct the address of the variable.

- **variable_name**

  A variable to be stored.

This macro stores a single byte of data present in a register to a memory location.

The following macro call:

```
ST_BYTE r3, r4, byte_var
```

is replaced with the following instructions:

```
addis r4, 0, ha(byte_var)
stb   r3, lo(byte_var)(r4)
```

ST_DOUBLE — Store Double-Precision

Syntax

ST_DOUBLE ***freg***, ***addr_reg***, ***variable_name***

Description

- **freg**

  A floating-point register to be stored.
- **addr_reg**

  A general-purpose register used to construct the address of the variable.
- **variable_name**

  A variable to be stored to.

This macro stores 8 bytes of double-precision data from a register to a memory location.

The following macro call:

```
ST_DOUBLE f0, r4, double_var
```

is replaced with the following instructions:

```
addis r4, 0, ha(double_var)
stfd  f0, lo(double_var)(r4)
```

ST_HWORD — Store Half Word

Syntax

ST_HWORD ***reg***, ***addr_reg***, ***variable_name***

Description

- **reg**

  A general-purpose register containing the half-word to be stored.

- **addr_reg**

  A general-purpose register used to construct the address of the variable.

- **variable_name**

  A variable to be stored to.

This macro stores 2 bytes of data to a memory location.

The following macro call:

```
ST_HWORD r3, r4, short_var
```

is replaced with the following instructions:

```
addis r4, 0, ha(short_var)
sth   r3, lo(short_var)(r4)
```

ST_SINGLE — Store Single Precision

Syntax

ST_SINGLE ***freg***, ***addr_reg***, ***variable_name***

Description

- **freg**

  A floating-point register to be stored.

- **addr_reg**

  A general-purpose register, used to construct the address of the variable.

- **variable_name**

  A variable to be stored to.

This macro stores 4 bytes of single-precision data to a memory location.

The following macro call:

```
ST_SINGLE f0, r4, float_var
```

is replaced with the following instructions:

```
addis r4, 0, ha(float_var)
stfs  f0, lo(float_var)(r4)
```

ST_WORD — Store Word

Syntax

ST_WORD ***reg***, ***addr_reg***, ***variable_name***

Description

- **reg**

  A general-purpose register containing the word to be stored.

- **addr_reg**

  A general-purpose register used to construct the address of the variable.

- **variable_name**

  A variable to be stored to.

This macro stores 4 bytes of data to a memory location.

The following macro call:

```
ST_WORD r3, r4, word_var
```

is replaced with the following instructions:

```
addis r4, 0, ha(word_var)
stw   r3, lo(word_var)(r4)
```

Chapter 9
Sample Assembler Session

During the first pass through the input source file, the assembler:

- Expands macros
- Examines labels, variables and user-defined symbols, and places them into the symbol table
- Decodes opcodes and directives and updates the assembly program counter

During the second pass, the assembler:

- Generates object code
- Resolves symbolic addresses
- Produces a listing and an object module

Errors detected during the assembly process are displayed in the output listing with a cumulative error count.

This chapter contains a description of a sample assembler listing created during the assembler's second pass.

Assembler Listing

The assembler can generate a listing including a symbol table or cross-reference table from the assembly information. The listing displays all information pertaining to the assembled program, both assembled data and your original source statements.

The main purpose of the listing is to convey all pertinent information about the assembled program, the memory addresses, and their values. You can use it as a documentation tool by including comments and remarks that describe the function of the particular program section. The relocatable object module, also produced during pass two, contains the object code address and value information in computer-readable format.

When error conditions are detected during the assembly process, a line titled **ERROR** appears just after the affected line of source code describing the errors in that line of code. At the end of the listing, the assembler prints the message **Errors:** *nn*, **Warnings:** *mm*. Warnings are represented by a **(W)** flag. Errors are represented by an **(E)** flag. An explanation of the individual assembler errors is in Appendix B, "Assembler Error Messages".

Figure 9-1 shows a sample output listing. Refer to the following numbered points in the figure in order to examine the listing.

1. The page headings on this sample show the time and date of the program run. This information might not be present in some installations.
2. The line titled **Command Line:** specifies the command line used to invoke the assembler.
3. The source listing contains the columns titled **Line** and **Address**.

   The **Line** column contains decimal numbers that are associated with the listing source lines. The cross-reference table refers to these numbers. Periods can be used to separate numbers. These periods can provide a distinction between nesting levels of included or macro-expanded code.

   The Address column contains a value that represents the first memory address of any object code generated by this statement or the value of a **.equ** or **.set** directive.

   In the sample listing, there are no errors or warnings. If the assembler detects error conditions during the assembly process, a line will be printed immediately after the line that produced the error indicating the name of the source file, the line number, an '(E)' designator, a number indicating the type of error, and then a textual description of the error. Warnings are represented with a '(W)' designator. At the end of the listing, the assembler prints the total number of errors and warnings encountered during the assembly of the source file.
4. The column titled **Opcode** contains object code generated by the assembly language source statement. The column appears to the right of the **Address** column. If the statement generates data for a data section, the first four bytes are printed. If the assembly statement is an instruction, the assembled object code is displayed.
5. Assembler relocation flags appear to the right of the data words:

   R Relocatable operand

   E External operand

   These letters are special indicators that mark assembly statements identified by the assembler to contain relocatable data. The absence of the indicator indicates no relocatable data is contained in the assembly statement.
6. Original source statements appear to the right of the above information in the column titled **Source**.
7. A cross-reference table appears at the end of the assembly listing. The table lists all symbols defined in alphabetical order, the section in which they were defined, and their final absolute values. Line numbers in which the labels occur are listed under **References**. The next section describes the cross-reference table in more detail.

Cross-Reference and Symbol Tables

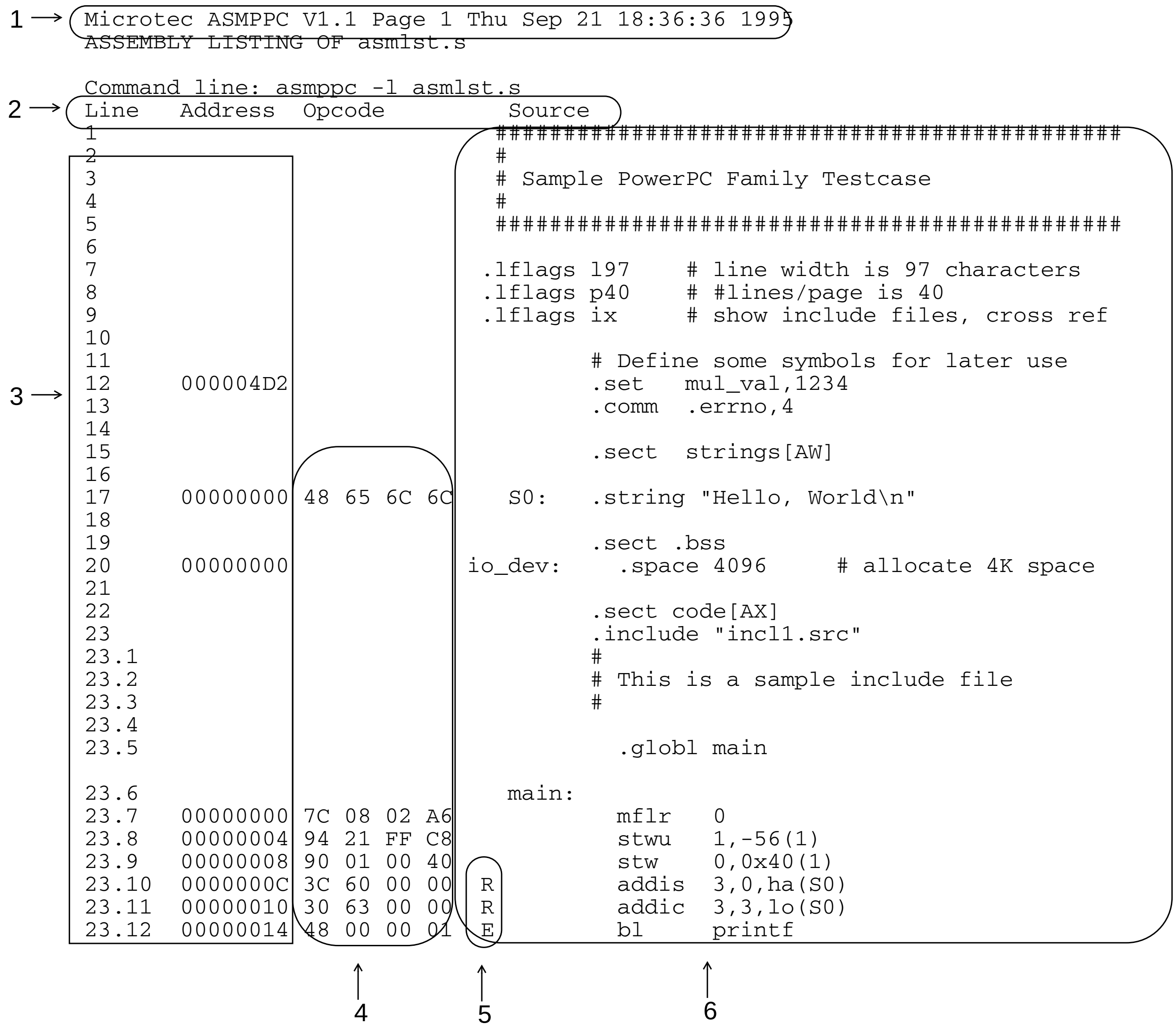
The cross-reference and symbol table options are turned off by default. To turn on the cross-reference table, use **.lflags x**. To turn on the symbol table, use **.lflags s**. The assembler will produce either a cross-reference table or a symbol table, but not both.

Figure 9-1 shows an example of the cross-reference output. Refer to the following points in order to examine the cross-reference table format.

1. All user-defined symbols in the program are listed under the heading **Label**.
2. A symbol's section, value, and any attributes are listed under the **Value** column.
3. Under **References**, a line number preceded by a minus sign (**-**) indicates that the symbol was defined on that line. Line numbers not preceded by a minus sign indicate a reference to a symbol. Note that for multiply-defined symbols, more than one definition can appear for the symbol.

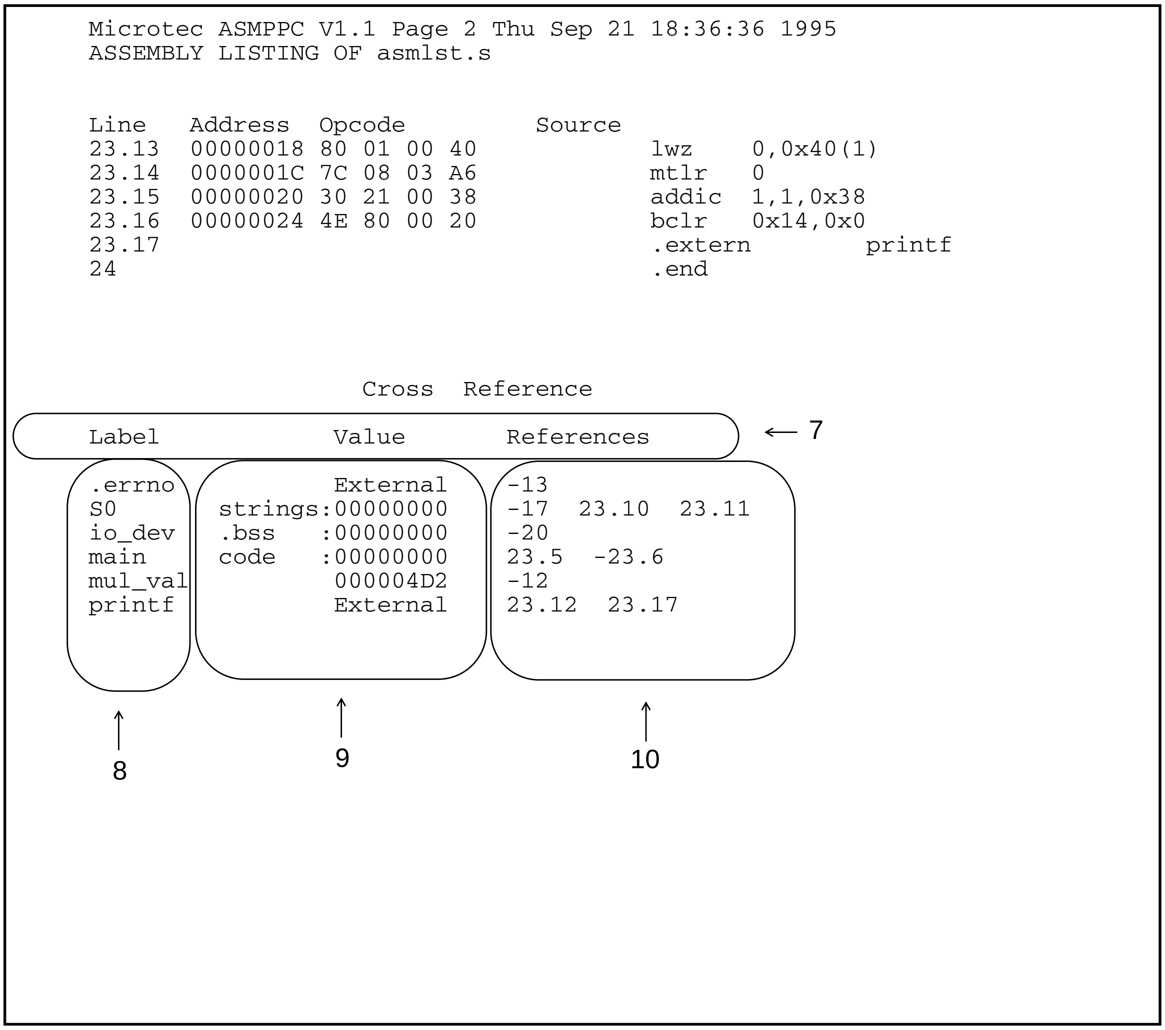
Section names and the module name symbols do not appear in the symbol table listing.

Figure 9-1. Sample Assembly Listing



```
Microtec ASMPPC V1.1 Page 1 Thu Sep 21 18:36:36 1995
ASSEMBLY LISTING OF asmlst.s

Command line: asmppc -l asmlst.s
Line   Address  Opcode         Source
1                             ##############################################
2                             #
3                             # Sample PowerPC Family Testcase
4                             #
5                             ##############################################
6
7                            .lflags l97    # line width is 97 characters
8                            .lflags p40    # #lines/page is 40
9                            .lflags ix     # show include files, cross ref
10
11                                   # Define some symbols for later use
12     000004D2                      .set   mul_val,1234
13                                   .comm  .errno,4
14
15                                   .sect  strings[AW]
16
17     00000000 48 65 6C 6C    S0:    .string "Hello, World\n"
18
19                                   .sect .bss
20     00000000            io_dev:    .space 4096     # allocate 4K space
21
22                                   .sect code[AX]
23                                   .include "incl1.src"
23.1                                 #
23.2                                 # This is a sample include file
23.3                                 #
23.4
23.5                                   .globl main

23.6                           main:
23.7   00000000 7C 08 02 A6              mflr   0
23.8   00000004 94 21 FF C8              stwu   1,-56(1)
23.9   00000008 90 01 00 40              stw    0,0x40(1)
23.10  0000000C 3C 60 00 00  R           addis  3,0,ha(S0)
23.11  00000010 30 63 00 00  R           addic  3,3,lo(S0)
23.12  00000014 48 00 00 01  E           bl     printf
```

Figure 9-2. Sample Assembly Listing (cont.)



```
Microtec ASMPPC V1.1 Page 2 Thu Sep 21 18:36:36 1995
ASSEMBLY LISTING OF asmlst.s


Line   Address  Opcode         Source
23.13  00000018 80 01 00 40            lwz    0,0x40(1)
23.14  0000001C 7C 08 03 A6            mtlr   0
23.15  00000020 30 21 00 38            addic  1,1,0x38
23.16  00000024 4E 80 00 20            bclr   0x14,0x0
23.17                                  .extern        printf
24                                     .end




                   Cross  Reference

Label            Value       References

.errno           External    -13
S0       strings:00000000    -17  23.10  23.11
io_dev   .bss   :00000000    -20
main     code   :00000000    23.5  -23.6
mul_val          000004D2    -12
printf           External    23.12  23.17
```

Chapter 10
Linker Operation

Many programs are too long to conveniently assemble as a single module. To avoid long assembly times or to reduce the required size of the assembler symbol table, long programs can be subdivided into smaller modules, assembled separately, and linked together by the LNKPPC Linker.

The primary functions of the linker are to:

- Resolve external references between modules and check for undefined references (linking)
- Adjust all relocatable addresses to the proper absolute addresses (loading)
- Output the final absolute object module

After the separate program modules are linked and loaded, the output module functions as if it had been generated by a single assembly.

When an absolute load is performed, relocatable addresses are transformed into absolute addresses, external references between modules are resolved, and the final absolute symbol value is substituted for each external symbol reference. The linker lets you specify program section addresses and external definitions. It also lets you assign the final load address and section loading order. Absolute output is produced in the Executable and Linking Format (ELF) format.

LNKPPC can also combine relocatable object modules into a single relocatable object module suitable for later relinking with other modules. This feature is known as incremental linking. In an incremental link, external references are resolved whenever possible. No section address assignment or resolution of relocatable references is performed. The relocatable output is an ELF relocatable file.

Linker Features

The LNKPPC Linker supports the following features:

- Output in ELF format
- Independently specified relocatable section load addresses
- Specified relocatable section loading order
- Incremental linking

- Values of previously defined global symbols modified at link time
- Loading of object modules from a library

  The linker will include only those modules from a library that are necessary to resolve external references. Library modules can also be used in incremental links.
- Definition of external symbols at link time, to force loading of modules from libraries
- Both Microtec IEEE-695 and System V PORTAR library file formats for input
- Inclusion of symbols and line number information in the absolute object module for symbolic debugging
- Cross-reference table and map file generation
- First-fit algorithm to place object modules in memory for better memory utilization
- Complex relocation
- Automatic copying of initialized data values, which can be placed in ROM
- Link-time verification of compatibility between different modules

  This check verifies that the resulting object does not contain instructions that will not execute appropriately in the targeted processor.
- Modification of global symbol names at link time
- Setting of section size
- MCCPPC generates information in the **.libinfo** section. This information is used by the linker to select a default libraries, and to check for compatibility between object files.

Program Sections

To effectively use the ASMPPC Assembler and LNKPPC Linker, you must understand sections and section attributes.

A section is a region of memory that contains information. There can be any number of general purpose sections that can contain both instructions and/or data. Each section contains its own location counter and typically is a logically distinct part of the total program. Instructions in one section can make a reference to any other section.

Sections are defined by the following attributes:

- Each section is identified by a symbolic name
- A section is relocatable
- A section has a type

- The components of a section can be aligned to a particular byte spacing
- There can be any number of general purpose sections

Different relocatable module sections (or subsections) with the same name and same type are combined into the same output section unless linker commands split the section. The combined section refers to the total code from all object modules that are associated with the section name. These combined sections form the output sections. The individual sections from different relocatable modules are the input sections. Input sections of the same name and same type are combined into one contiguous output section.

The subsections of an output section are loaded into a contiguous block of memory and do not overlap. The size of a section is the sum of the sizes of all its subsections, but must be less than four gigabytes. Sections have a type attribute and an alignment attribute.

Relocatable Sections

Relocatable sections do not have a specific load address associated with them while being input to the linker. Relocatable sections are given addresses in memory during an absolute link.

Section Alignment

The beginning address of each file’s contribution to a section is aligned by default at word boundaries for text sections and byte boundaries for data sections. Also, section alignment can be modified at assembly time by either the section definition or the section contents (word or double-word sized data). This alignment can be increased by using the linker **ALIGN** command. Specified absolute addresses will be rounded up to the next aligned word boundary if they do not fall on one.

The section alignment attribute can be any power of two. The section alignment attribute affects the beginning address of each file’s contribution to a section (that is, a subsection). That is, if several files each define a relocatable section named **A** and the alignment for this section is four, then the beginning address of section **A** in each file will be rounded up to a modulo four boundary if necessary. For more information on section alignment, see the **ALIGN** linker command.

Section Name

Section names, like all names in the linker, cannot exceed 8192 characters in length or cannot contain the following special characters:

*	asterisk	{	left brace	]	right bracket
:	colon	[	left bracket	)	right parenthesis
,	comma	(	left parenthesis	;	semicolon

=	equal sign	\n	newline		space
!	exclamation	}	right brace	\t	tab

Section Type

The section type attribute shows what a section contains. Program code sections generally contain instructions. Data sections generally contain read/write data items. ROMable data sections generally contain read-only data items.

The section type indicates whether or not allocation and relocation of a section will take place. In the following discussion of section types, “allocated” means that a virtual address and nonzero size have been assigned to the section. “Relocated” means that the virtual address is the section’s absolute memory address in the image and that all relocations have been resolved within that section.

Sections can be ordered by type using the **ORDER** command. Table 10-1 defines the section types.

Table 10-1. Section Types

Section Name	**Meaning**	**Section Type**	**ELF Section Attributes**
.bss	Uninitialized data	BSS	ALLOC, WRITE
.sbss, .sbss2, .PPC.EMB.sbss0	Uninitialized data	BSS	ALLOC, WRITE
.sdata, .sdata2, .PPC.EMB.sdata0	Data	DATA	ALLOC, WRITE
.data, .data1	Data	DATA	ALLOC, WRITE
.debug_abbrev, .debug_aranges, .debug_frame, .debug_info, .debug_line, .debug_loc, .debug_macinfo, .debug_pubnames, .debug_str	DWARF Version 2 debug information	DWARF2	Not applicable
.PPC.EMB.procflags	Processor information	LIT	ALLOC
.rodata, .rodata1	Read-Only Data	LIT	ALLOC
.rel*name* or .rela*name*	Information to relocate *name*	RELOC	Not applicable
.text, .init, .fini	Program code	TEXT	ALLOC, EXEC

Table 10-1. Section Types (cont.)

Section Name	**Meaning**	**Section Type**	**ELF Section Attributes**
tool_dependent	Non-executable text that is placed in the output file	NOTE	Not applicable

BSS Section Type

Sections in the **BSS** section type contain space for uninitialized variables. The absolute address of the **bss** section immediately follows the data section unless overridden by commands or located in a “hole” between sections. A **bss** section does not contain relocation entries, line numbers, or data. The section has a section entry and a size, but actually occupies no space in the object file (although it can influence the padding requirements for the other sections). Symbols for variables within the section and symbols that refer to the section are relocated. It is presumed that the contents of all BSS sections will be initialized to 0 by the program’s startup code.

DATA Section Type

Sections in the **DATA** section type contain the initialized constant and variable data of a program. The default action of the linker is to combine all data sections of the same name into one output section of that name and locate the combined data section immediately after the literal section(s).

DWARF2 Section Type

Sections in the **DWARF2** section type contain all the symbolic and line number debugging information in the **DWARF** Version 2 format. ELF sections whose names start with **.debug** are included in this section type. These sections are not allocated and do not take any memory space. The sections are stored in the output file.

NOTE Section Type

Sections in the **NOTE** section type are information sections. They are ASCII text that contain information about the object file. Memory is not allocated for this section type, and these sections are not relocatable. They are used for adding version or copyright information to the output file.

LIT Section Type

Sections in the **LIT** section type are literal sections. They contain read-only data. The default action is to combine all literal sections of the same name into one output section of that name. The combined section is then placed immediately after the text section(s).

RELOC Section Type

Sections in the **RELOC** section type contain the relocation information for the sections in the module. Sections with names **.rel***name* or **.rela***name*, where *name* is the name of the section to which the relocation information refers, will be included in this section type. These sections are not allocated and do not take any memory space. The sections are stored in the output files during incremental linking.

TEXT Section Type

Sections in the **TEXT** section type contain the normal instruction space of a program. The default action of the linker is to combine all text sections of the same name into one output section of that name. The combined text section is then placed as the first output section in the file.

Special Sections

In a PowerPC executable file, there may be some special sections called Small Data Area sections. Each Small Data Area has a base value that is used to address these special sections. The special sections **.sdata** and **.sbss** use the address represented by the symbol **_SDA_BASE_** as their base, while the special sections **.sdata2** and **.sbss2** use the address represented by the symbol **_SDA2_BASE_** as their base. For the sections **.PPC.EMB.sdata0** and **.PPC.EMB.sbss0**, the value of 0 (zero) serves as the base value. The sections **.sdata**, **.sdata2**, and **.PPC.EMB.sdata0** hold initialized data that contribute to the program memory image. The special sections **.sbss**, **.sbss2**, and **.PPC.EMB.sbss0** are intended to hold writable data that are initialized to zero. Up to 64 kilobytes of data can be addressable from each base symbol.

If an executable file contains Small Data Area sections, the linker sets their corresponding base symbols to a value such that the special sections can be addressed with a 16-bit signed offset from that base value. For the sections **.PPC.EMB.sdata0** and **.PPC.EMB.sbss0**, the base value is always zero.

During linking, if the value of the base of a Small Data Area is not set by the user, the linker calculates an address to serve as the value of the base, from which the pair of Small Data Area sections can be addressed using a 16-bit signed offset. If the user supplies the value of any Small Data Area bases through global **.equ** symbols or **PUBLIC** commands, then the linker tries to allocate the corresponding sections in memory in such a way that these sections can be addressed from the base address using a 16-bit signed offset. If the linker fails to allocate the special sections in such a manner, an error is issued, and the link is aborted.

The **.PPC.EMB.procflags** special section contains information that assists tools that consume the absolute object file. This information identifies the target processor that the object file was created to execute on. It also contains information regarding invalid instruction contents, incomplete processor information, and the types of instructions that were used in input object files. The information contained in this section is summarized as part of the header in the map file.

The linker will use information contained in the input object files to determine which target processor the output object file may be executed on. If a program only contains instructions that are generic in nature, then the resulting code can be used on any target processor. However, if the program contains instructions that are meant for only a particular PowerPC processor variant, then the linker will indicate in the map file which target this program can safely be executed on.

Likewise, the linker will not allow the mixing of input objects that are not compatible with each other. If an input object that contains particular instructions for one PowerPC processor variant is linked to another input object that contains instructions that are valid only on a different PowerPC processor variant, then the linker will give an error message indicating that this has occurred.

Finally, some PowerPC processor variants are related to each other and may be safely linked together. For example, input objects compiled or assembled for the 603, 603E (PID 6), 603E (PID 7), EC603E (PID 6), and EC603E (PID 7) processor variants may be safely linked together. Also, input objects meant for the 403GA, 403GB, 403GC, and 403GCX processor variants, or the 604 and 604E processor variants, or the 740 and 750 processor variants may be linked together. The linker map file will indicate which variant the resulting executable can be safely executed on. If an EC603E variant is linked to a 603E variant, the resulting variant will be the 603E variant, with a PID value that is the highest of the two processor variants (PID 6 or PID 7).

The **-v** command line option can be used to generate a complete list of processor variants that may be safely linked together. This option lists each PowerPC processor variant and then gives a list of variants that are either subsets (that is, have an instruction and register set contained entirely within the current processor variant) or a superset (that is, have additional instructions and/or registers). If a variant is linked with a subset variant, the resulting variant type will continue to be the current variant. If a variant is linked with a superset variant, the resulting type will be the superset. In some cases, linking two variants together results in a different processor variant that is a superset of the two variants. The linker issues a warning if an attempt is made to link object files for two incompatible processor variants together.

Memory Space Assignment

You can control both the order in which sections are assigned space in memory and the initial address (load address) of any or all sections. Specifying the load address of a section does not alter the order in which sections are assigned space, but it affects the location in memory of subsequent sections that do not have load addresses specified.

The following kinds of addresses are used in this manual:

- A load address is the memory address at which the lowest byte of a section is placed.
- A base address is the lowest address considered for loading relocatable sections of the absolute object module after handling reserved memory, if any.

- A starting address is the location at which execution begins.

The order in which sections are assigned memory is as follows:

1. Sections named in the **ORDER** commands, in the sequence in which they are listed.
2. All available Small Data Area sections that were not specified through an **ORDER** command in the following order: **.PPC.EMB.sdata0**, **.PPC.EMB.sbss0**, **.sdata**, **.sbss**, **.sdata2**, and **.sbss2**.
3. All other sections are loaded in the sequence in which their names were encountered by the linker.

While assigning memory for the Small Data Area sections, the linker first determines if the corresponding base value has been specified by the user. If such a value is specified, then the linker tries to place the sections in memory in such a way that the contents of the sections can be addressed from the base value using a signed 16-bit offset. If the linker cannot find a location within the 64K region that is addressable from the base value, the linker issues an error and aborts. For the sections **.PPC.EMB.sdata0** and **.PPC.EMB.sbss0**, the linker tries to place these sections so they can be addressable from address 0 using a 16-bit signed offset. If a base value is not supplied by the user, the linker assigns memory to the Small Data Area sections and calculates the corresponding base value to be the midpoint of the lowest and highest address occupied by the corresponding sections. After calculating the base value, if the corresponding Small Data Area sections are too large to be addressed with a 16-bit signed offset, an error is issued, and the link aborts.

The linker encounters a section name when the name appears in a command or when a module that refers to that section name is loaded.

Note

Library relocatable object modules that are not selected for inclusion in the absolute object module do not have their section names examined by the linker.

To assign memory to a section, assign it a load address. You can specify section load addresses in a linker **ORDER** command. Otherwise, the linker assigns load addresses as follows:

1. The load address of the first input section is assigned the base address, where the default base address is 0.
2. The load address of each subsequent input section is assigned immediately above the preceding input section at an aligned boundary.

Relocation Types

Sections can have zero or more relocation entries in the object file, each referring to a symbol in the section that the linker will relocate. These relocation entries will be found in associated ELF sections of type **RELOC**. The name of the associated **RELOC** section is either **.rel***name* or

.rela*name*, where *name* is the name of the section. A relocation entry has three fields: an offset, a relocation type, and a symbol index. If the section containing the relocation entries has a name beginning with **.rela**, an additional field, the **addend**, is included in each relocation entry. When present, this constant addend will be added into the relocation calculation, along with the symbol value. When the linker relocates a section, it processes all relocation entries for that section, using the offset to find the data or instruction in the section to which to apply the relocation. Using the symbol table index, the linker obtains the symbol’s value and adjusts the address to the proper absolute address according to the relocation type and addend, if present. Absolute symbols are not adjusted. The result of the calculation is then stored at the indicated offset.

Incremental Linking

The linker can produce a single relocatable object module from assembled relocatable object modules through the process of incremental linking. The linker resolves all external references between the modules loaded and allows undefined external references to other modules to exist in the output object module. The external references are reported on the link map.

Incremental linking is useful as it lets groups of users easily share relocatable object modules for the joint development of code. Long lists of previously checked object modules do not have to be linked with those modules currently under development. Only one incrementally linked module has to be linked, making it unnecessary to know all the original module names that are being linked with the new code.

The following example shows incremental linking of four test modules: **TEST1**, **TEST2**, **TEST3**, and **TEST4**. In the first part of the example, a normal load is performed requiring four load commands. In the second part of the example, three of the modules are incrementally linked into **TEST123**. **TEST123** and **TEST4** are then linked to produce one absolute output file.

Example 10-1. Incremental Linking

```
;TEST.CMD; A command file that consists of four
      ; load commands to link four relocatable
      ; object modules.
LOAD TEST1
LOAD TEST2
LOAD TEST3
LOAD TEST4
```

The output object module produced is **TEST.x**.

The same result could be achieved with the two following load command sequences:

```
;TEST123.CMD; A command file consisting of three
         ; load commands to link three
         ; relocatable object modules.
LOAD TEST1
LOAD TEST2
LOAD TEST3
```

The output object module produced is **TEST123.o**.

```
;TESTF.CMD; A command file consisting of two
         ; load commands to link two
         ; relocatable object modules, one of
         ; which is a combination of
         ; previously linked modules.
LOAD TEST123.o; for UNIX only
LOAD TEST4
```

The output object module produced is **TESTF.x**.

Link-Time Definable Externals and Entry-Point Name

Unresolved external symbols can be created at link time through a command line option. An unresolved external will force an initial search of libraries until the symbol is found. If the symbol remains unresolved, a link error will occur.

The entry-point name specifies where to set the program counter at execution time. The entry point can be entered on the command line, which is useful when loading stand-alone programs. The default entry point name for the LNKPPC Linker is **__mri_start**. If the entry-point is not found, the linker sets the entry-point address to zero. The **START** linker command can also be used to specify the entry point.

Linker-Generated Symbols

Some symbols in object modules are meaningful to the linker. Likewise, the linker creates some symbols, which may be used in programs or for debugging purposes.

As was mentioned in the “Special Sections” section earlier in this chapter, the linker supports three Small Data Areas. One of the 64k regions is already mapped around address 0. For the other two regions, though, the linker recognizes the two global symbols _SDA_BASE_ and _SDA2_BASE_. These symbols represent a midpoint in a Small Data Area and its value is meant to be loaded into an associated PowerPC general register (r13 and r2, respectively). If these global symbols are defined with a constant value by the user, the linker will attempt to map the associated sections into the 64k region that is represented by that value. If the global symbols do not exist, the linker will place the associated sections in close proximity to each other (if allowed to) and will create the global symbol with a value that can address all of the section's contents. If that is not possible, an error is emitted. If none of the associated sections exist and the global symbols are not defined by the user, the linker will create the symbols with a value of 0.

Another global symbol that can be created by the linker is the **__initdat** symbol. This symbol is created when the **INITDATA** command is used. The symbol is given a value that represents the address of the first byte in the linker-created **initdat** section, which holds a representation of the sections used with the **INITDATA** command. Code can reference the **__initdat** symbol to begin reading the contents of that section.

Some other global symbols are created by the linker for use with some operating systems. These symbols are only created if a global symbol with the same name does not already exist. The **_etext** symbol is given a value that is one byte past the end of the **.text** section. The **_edata** symbol is given a value that is one byte past the end of the **.data** section. The **_end** symbol is given a value that is one byte past the end of the **.bss** section. If those sections do not exist, the global symbol is created with a value of 0.

Finally, there are symbols that have special meaning to the linker. These symbols are generally just checked for and are not created by the linker (with one exception).

In many cases, it is useful to know how big a section is. The assembler has the **SIZEOF** operator, which returns the size of a section. This operator works by creating a reference to an external symbol that is the name of the section, with **.sizeof** appended to its name. When the linker sees this external symbol, it creates a global symbol of the same name whose value is the size of the section. This will not work correctly if there are multiple sections with the same name, but different types, or if the user splits the section up into multiple pieces with the **ORDER** directive. In those situations, the linker will give an error indicating that an appropriate value cannot be assigned to the symbol. The **SIZEHA**, **SIZEHI**, and **SIZELO** operators function in a similar manner, returning the high adjusted 16 bits, high 16 bits and low 16 bits of the size of the section, respectively.

Sometimes, the starting address of a section is required. The assembler has the **SECTOF** and **STARTOF** operators, which return the address of the start of the section that is its argument. The **STARTOF** operator works by creating a reference to an external symbol that is the name of the section, with **.startof** appended to its name. When the linker sees this external symbol, it creates a global symbol of the same name whose value is the starting address of the section. The case with **STARTHA**, **STARTHI,** and **STARTLO** operators function in a similar manner, returning the high adjusted 16 bits, high 16 bits and low 16 bits of the address of the start of the section, respectively.

Another concern that the linker has is that the program being created should execute correctly on the target PowerPC processor. Some examples of problems that can occur is that instructions could be linked into the program that will not execute on the target processor, or that a module being linked in contains errors. This information is passed to the linker through the **._PPC_EMB_procflags** local symbol. The assembler creates this symbol with appropriate information to indicate the status and contents of an object file. The linker can then use the value of this symbol to determine if the link can be safely performed. If there are problems, the linker can signal them to the user. The **._PPC_EMB_procflags** symbol is modified by the linker and, in a final link, is placed in the **.PPC.EMB.procflags** section.

Another symbol that the linker looks for is **._PPC_EMB_sim_present**. This global symbol is used to communicate to the linker that a module has been linked in that performs floating-point (and possibly other) instruction simulation. This is important for some processors that do not have a floating-point unit. If a program containing floating-point instructions is executed on such a processor, a trap will occur. If no simulator is present at the trap address, then the program will halt or definitely not behave correctly. Since the linker knows about the contents of Microtec object files that are being linked, it can detect when floating-point or other instructions exist that cannot execute on the target processor. If the **._PPC_EMB_sim_present** global symbol is not present, then that tells the linker that no simulator module has been linked in and the linker gives an error to indicate that execution problems may occur. It should be noted that this error can be ignored by simply defining the **._PPC_EMB_sim_present** symbol with a PUBLIC command and any value. This may be useful if floating-point simulation is available through a routine in ROM that is not being linked into this application.

The final local symbols involve the C++ **-KLi** flag. Since different flag values can produce object modules that cannot be linked together, the assembler passes information to the linker so it can validate them. This is done through the **._PPC_EMB_initflags** and **._PPC_EMB_initrn_** symbols. If mismatched **-KLi** settings are detected by the linker, an error will indicate which modules need to be recompiled.

Map File

The map file indicates how sections are allocated in memory. It includes the linker command file, output module information, a summary of the section information, summaries of any merged regions (such as templates or inlined functions), a global symbol table, and a cross-reference table. The merged region summary shows which instance of a given region was retained and can also indicate which regions were removed. The global symbol table shows the absolute locations of each global symbol. The cross-reference table listing includes all the module names that reference each global symbol. Command line flag options control which tables appear in the map file.

Chapter 12, “Sample Linker Session”, lists and describes an example map file.

Symbol Name Length

The LNKPPC Linker map file displays symbol names up to 8192 characters. If a symbol is longer than 14 characters, the LNKPPC Linker places the symbol on a line by itself. The **SECTION**, **ADDRESS**, and **MODULE** information start in the corresponding columns of the next line following the long symbol name.

Cross-Linking Between Processor Variants

There are many different types of PowerPC processor variants. Some of these variants represent a family of related chips, where each member is a superset of the functionality of a related member in that family (for example, the 603 family). Other variants represent a capability that is similar to other PowerPC processor variants but that contains enough differences that code that runs on one variant cannot safely be executed on a different variant (for example, the 401GF). Mentor Graphics tools keep track of the contents of various object files and ensure that only compatible object files are linked together.

This check is only performed if two object files were compiled for different PowerPC processor variants. No check is necessary if two object files compiled for the same chip are linked together. Similarly, if an object file contains only generic instructions or SPR references, then it is generally considered safe to link that object file with any other object file. The resulting type will be the same as that of the first object file in the link that contains processor-specific instructions or SPR references. If a link containing incompatibilities between processors is detected, the linker will emit an error.

The presence of floating-point instructions in the input object files is detected through the same mechanisms. The linker will use the final PowerPC processor variant type to determine whether a simulation library is required or not. The presence of floating-point instructions will not result in an inability to link two object files together.

Table 10-2 shows some results of linking object files compiled for various processors with other object files.

Table 10-2. PowerPC Variant Cross-Link Compatibility

Object file variant	Resulting variant from previous objects	New object variant
603	603	603
	603E (603E6)	603E (603E6)
	603E7V	603E7V
	EC603E (EC603E6)	603E (603E6)
	EC603E7V	603E7V
603E (603E6)	603	603E (603E6)
	603E (603E6)	603E (603E6)
	603E7V	603E7V
	EC603E (EC603E6)	603E (603E6)
	EC603E7V	603E7V
603E7V	603	603E7V
	603E (603E6)	603E7V
	603E7V	603E7V
	EC603E (EC603E6)	603E7V
	EC603E7V	603E7V

Table 10-2. PowerPC Variant Cross-Link Compatibility (cont.)

Object file variant	Resulting variant from previous objects	New object variant
EC603E (EC 603E6)	603	603E (603E6)
	603E (603E6)	603E (603E6)
	603E7V	603E7V
	EC603E (EC603E6)	EC603E (EC603E6)
	EC603E7V	EC603E7V
EC603E7V	603	603E7
	603E (603E6)	603E7
	603E7V	603E7
	EC603E (EC603E6)	EC603E7
	EC603E7V	EC603E7
604	604	604
	604E	604E
604E	604	604E
	604E	604E
403GA	403GA	403GA
	403GB	403GA
	403GC	403GC
	403GCX	403GCX
403GB	403GA	403GA
	403GB	403GB
	403GC	403GC
	403GCX	403GCX
403GC	403GA	403GC
	403GB	403GC
	403GC	403GC
	403GCX	403GCX

Table 10-2. PowerPC Variant Cross-Link Compatibility (cont.)

Object file variant	Resulting variant from previous objects	New object variant
403GCX	403GA	403GCX
	403GB	403GCX
	403GC	403GCX
	403GCX	403GCX

The **-v** command line option can be used to generate a complete list of processor variants that may be safely linked together. This option lists each PowerPC processor variant and then gives a list of variants that are either subsets (that is, have an instruction and register set contained entirely within the current processor variant) or a superset (that is, have additional instructions and/or registers). If a variant is linked with a subset variant, the resulting variant type will continue to be the current variant. If a variant is linked with a superset variant, the resulting type will be the superset. In some cases, linking two variants together results in a different processor variant that is a superset of the two variants. The linker issues a warning if an attempt is made to link object files for two incompatible processor variants together.

While many of the variants support identical instruction and SPR sets, they have differences with respect to memory-mapped registers, peripherals, the amount of on-chip memory, or other issues that are not tracked at the compiler or assembler level. Linking such object files together would not produce code that would necessarily work for any particular PowerPC processor.

Chapter 11
Linker Commands

The LNKPPC Linker reads a sequence of commands from a specified command file and/or an interpreted command line. This chapter provides reference information about the available linker commands and their syntax. For more information on command line and command file usage, see Chapter 2, “Command Usage”.

Command Syntax

Commands and command arguments can begin in any column. Command arguments must be separated from the command by at least one blank. Only one command is permitted per line. Comments are indicated by a semicolon (**;**) and are terminated with a newline character.

Statements can be continued by specifying a backslash (\) at the end of a line, up to a maximum of 32,768 characters per statement.

Table 11-1 shows how numeric command arguments can be represented.

Table 11-1. Representation of Numeric Command Arguments

Type	**Indicator**
Hexadecimal	Preceded by **0x** or **0X**
Binary	Preceded by **0b or 0B**
Octal	Preceded by **0**
Decimal	None

If a numeric command argument is not explicitly described using these methods, the linker assumes the argument to be decimal. Using multiple base indicators on the same value is illegal.

Symbols

Symbols and section names must follow the syntax rules for symbols given in the assembler manual. The first character in a symbol must be alphabetic, a period (.), or an underscore (_). Subsequent characters must be alphabetic, numeric, a period, or an underscore. Symbols are limited to 8192 characters in length.

Case Sensitivity

Section names, symbols, and reserved names (such as **TEXT**) are case-sensitive by default in the linker command file.

Linker Commands

The linker commands are listed alphabetically in Table 11-2, and each command is discussed in detail in the subsequent pages. Note that during an incremental link, only the **INCLUDE**, **LISTMAP**, and **LOAD** commands are valid.

Table 11-2. Linker Commands

Command	**Function**
ABSOLUTE	Specifies sections to be emitted
ALIGN	Aligns an output section to a specified boundary
EXTERN	Creates external references
INCLUDE	Includes the text of a command file into another command file
INITDATA	Creates ROM section for RAM initialization
LISTMAP	Controls presence of tables in the map file
LOAD	Loads object modules
ORDER	Specifies section order
PUBLIC	Reassigns global symbol value
RESADD RESNUM	Reserves memory
SECTSIZE	Sets the maximum section size
START	Specifies entry point
SYMTRAN	Modifies global symbol names

ABSOLUTE — Specifies Sections to be Emitted

Syntax

ABSOLUTE ***section***[,*section*]...

Description

- **section**

  {*section_name*[!*section_type*] | !*section_type*}

  where:

section_name	Indicates the name of the output section.
section_type	Indicates section type. The only valid types are **TEXT**, **LIT**, **DATA**, and **BSS**.

The **ABSOLUTE** command specifies which sections to generate in the absolute output module. In the absence of any **ABSOLUTE** commands, the linker emits all sections into the output object module.

If *section_type* is omitted from *section*, then all sections having the given name will be loaded in the order that they appear from the input and not in the order that they appear in this command.

Although the section combining rules specifies that sections are to be combined on name and type, the default output order is based strictly on type. Specifically, all sections of type **TEXT** will be output first. Then, all sections of type **LIT** will be output, followed by all sections of type **DATA**, and ending with sections of type **BSS**. If this directive is supplied, all output sections specified will be loaded and no other sections encountered will be generated.

This directive applies to output sections only. Therefore, specifying input section names or individual input module names on this directive are not permitted.

Notes

This command is useful for isolating certain portions of the output file that will be burned into PROMs.

Example

```
ABSOLUTE .text
```

In this example, all output sections named **.text** are placed in the output file. If the linker encounters other loadable sections, such as **.data** or **.rodata**, their section contents will not be written to the output file, although they will be allocated and relocated as if they were to be emitted.

ALIGN — Aligns an Output Section to a Specified Boundary

Syntax

ALIGN [*section*]=***value***[,[*section*]=*value*]...

Description

- **section**

  {*section_name*[!*section_type*] | !*section_type*}

  where:

section_name	Indicates the name of the output section.
section_type	Indicates section type. The only valid types are **TEXT**, **LIT**, **DATA**, and **BSS**.

- **value**

  Specifies the boundary to be used for alignment. *value* is a number that is a positive integral power of two.

The **ALIGN** directive aligns an output section to a specified boundary, as indicated by *value*. The section will begin on the specified boundary. By default, the linker uses the largest align value of all input sections being combined into the output section. For linker generated sections, the alignment is four for text sections and one for data sections. This directive can specify an alignment requirement for one section or for all sections of the output file.

Multiple **ALIGN** directives are applied sequentially. If a given section is indicated in multiple **ALIGN** directives, either through basic section type specifications or default actions, the last **ALIGN** directive that applies to that section will specify the section's alignment.

Examples

```
ALIGN .data!DATA=16
```

In this example, the combined output **.data** section will begin on a 16-byte boundary. Any following output sections will follow the default alignment boundary.

```
ALIGN =16
```

In this example, all output sections will begin on a 16-byte boundary.

```
ALIGN .rodata!DATA=256
ALIGN .rodata1!DATA=256
ALIGN =4
```

The first two **ALIGN** directives set the **.rodata** and **.rodata1** sections to 256-byte boundaries. However, the next **ALIGN** directive changes the alignment for all sections to a 4-byte boundary, including **.rodata** and **.rodata1**. In order to keep the setting of 256 for **.rodata** and **.rodata1**, they must be listed after the general **ALIGN** directive.

EXTERN — Creates External References

Syntax

EXTERN ***name***[,*name*]...

Description

- **name**

  Indicates the symbolic name of an external reference.

The **EXTERN** command creates external references for the linker to resolve. You can create these external references by using the **EXTERN** command to add an entry for each named symbol to the linker's internal symbol table. Information indicating that these are external symbols to be resolved is inserted with the entry into the symbol table. If the symbol already exists in the symbol table either as an external reference or definition, the **EXTERN** command is ignored, and a warning is issued.

The **EXTERN** command can appear anywhere in a command file. Multiple **EXTERN** commands are allowed.

The **EXTERN** command is in effect for a given name when that name is specified in the command. An **EXTERN** command with a specific name can appear anywhere in a command file with respect to the **LOAD** command for the library in which the specific external symbol is defined. However, if the **EXTERN** command appears after the **LOAD** command, the module will not be loaded until a later pass is made through the libraries while trying to find definitions for any remaining unresolved external symbols. So, if a different library is specified in a **LOAD** command after the **EXTERN** command that also has a definition for the same symbol, the module from that library will be loaded, since it is the first definition that can be found that resolves a given external symbol. For that reason, it may be better to define the external symbol before the **LOAD** command that will contain the definition for that symbol.

Notes

The **-u** *name* command line option has the same effect as inserting an **EXTERN** command into the command file before the first **LOAD** command, if any.

Example

```
EXTERN g1
LOAD module1.o,module2.o,textern.lib
END
```

In this example, the symbol **g1** is not defined in both **module1.o** and **module2.o**. So if a definition of **g1** exists in **textern.lib**, the appropriate module that contains the definition will be force-loaded in order to resolve the external reference.

INCLUDE — Includes the Text of a Command File Into Another Command File

Syntax

INCLUDE ***filename***

Description

- **filename**

  Indicates the name of a command file whose contents are to be processed as if they were part of the current command file. This name can be either an absolute path or a relative path from the current directory.

The **INCLUDE** command causes the indicated file to be processed as if its contents were part of the current command file. The commands of the include file will be processed and executed as if a single command file were used. Include files can be nested up to 16 levels. Any attempt to include files beyond the maximum nesting level will result in a warning message and the invalid **INCLUDE** command will be ignored.

In one case, using an include file is not the same as having a single command file. If there are object files or libraries listed on the command line, the linker artificially inserts **LOAD** commands into the command file so those files are processed. This insertion occurs before any **LOAD** commands that are encountered in the command file. If no **LOAD** commands exist in the command file, the **LOAD** commands are added to the end of the command file. The linker does not check include files for the presence of **LOAD** commands; because of this, replacing an **INCLUDE** command with the contents of the include file could potentially result in different behavior from the linker.

Example

```
INCLUDE t1.inc
```

The contents of the file **t1.inc** will be processed by the linker. The results of this operation will be the same as if the commands in that file were part of the current command file at the location of the **INCLUDE** command.

INITDATA — Creates ROM Section for RAM Initialization

Syntax

INITDATA ***section***[,*section*]...

Description

- **section**

  {*section_name* [!\[*section_type*] [!*object_file*]]
  |!*section_type*[!*object_file*] |!!*object_file*}

section_name	Indicates the name of the section.
section_type	Indicates section type. The only valid types are **TEXT**, **LIT**, **DATA**, and **BSS**.
object_file	Indicates the input object file name where the section exists.

The **INITDATA** command, in conjunction with the C library routine **initcopy**, allows data to be copied from ROM to RAM during initialization. This feature is intended for use with embedded applications that have initialized values for data, which means that the data must be stored in ROM. For these applications, it is often necessary to copy the initialized values for all variables from a ROM section to a RAM section where the variables will reside and be modified while the application is executing.

INITDATA automatically creates a new **LIT** section in ROM named **initdat** and fills that section with a copy of all initialized data values contained in the section(s) named as command arguments. In order to facilitate the copying mechanism at run time, the linker also generates an external symbol named **_ _ initdat**, which contains the starting absolute address of section **initdat**.

Multiple section names may be given in the **INITDATA** command. The **ORDER** command can be used to specify absolute addresses for both the **initdat** section (usually in ROM) and the destination **DATA** sections (usually in RAM). If no such absolute address is supplied for **initdat**, the linker will allocate memory for that section last.

The exclamation point (**!**) character must be followed by an argument.

Example

```
INITDATA sec1,sec3,sec4
```

The previous example will cause the linker to create a **initdat** section at link time containing a copy of the contents of sections **sec1**, **sec3**, and **sec4**.

LISTMAP — Controls Presence of Tables in the Map File

Syntax

LISTMAP [[NO]PUBLICS [,] [NO]LOCALS [,] [NO]CROSSREFS [,] [NO]RETAINED]

Description

- ***PUBLICS***

  Indicates that the global symbol table is to be emitted into the map file. The table can be omitted by specifying **NOPUBLICS**.

- ***LOCALS***

  Indicates that the local symbol table is to be emitted into the map file. The table can be omitted by specifying **NOLOCALS**.

- ***CROSSREFS***

  Indicates that the symbol cross-reference table is to be emitted into the map file. The table can be omitted by specifying **NOCROSSREFS**.

- ***RETAINED***

  Indicates that the retained symbol table is to be emitted into the map file. This table can be omitted by specifying **NORETAINED**.

The **LISTMAP** command controls the presence or absence of the global symbol, local symbol, symbol cross-reference, and retained symbol tables in the non-XML map file. The default linker behavior is to emit all of these tables.

To suppress the emission of one or more of these tables, add the prefix “**NO**” to the name of the table that is not desired. The absence of the “**NO**” prefix will cause the indicated table to be emitted into the map file. If more than one table is to be specified, then the table names must be separated by either white space or commas on the command line. All arguments are case-insensitive and the order of the arguments does not matter. If the same table is indicated more than once, the last argument takes precedence.

Notes

The **LISTMAP** command is only meaningful if a non-XML map file is produced. Otherwise, it has no effect.

Example

```
LISTMAP NOLOCALS , PUBLICS
```

In this example, the local symbol table will not be emitted into the map file, if a map file is produced. The global symbol table will be emitted. The symbol cross-reference table will be emitted since that is the linker’s default behavior.

LOAD — Loads Object Modules

Syntax

LOAD ***module*** [*,module*]...

Description

- **module**

  Specifies a file containing the object module. Input object modules can consist of relocatable modules from the assembly process, relocatable modules from incremental linking, or libraries.

The **LOAD** command specifies one or more input object modules to be loaded or searched for declarations of unresolved externals. The linker differentiates between input modules on the basis of their internal format. When a library name is encountered, it will be searched only if unresolved externals remain and only those modules resolving externals will be loaded. The library can be either System V PORTAR or Microtec IEEE-695 library format.

Libraries are searched in the order found. Therefore, modules from a library will only be loaded if there are unresolved external symbols at that time. However, the linker will continue to search libraries in the order that they were specified until no more external symbols remain or all symbols are resolved by loading modules from the libraries. This means that libraries only need to be specified on the command line once. However, if a global symbol in a library is referenced after that library is loaded, then the module from that library will not be loaded until all remaining input object files and libraries are loaded. So it is possible that a global symbol from a later module or library might be used to resolve that external symbol, if more than one input library contains the same global symbol.

Incremental linking accepts relocatable modules produced from the assembly and produces a single relocatable object module. See Chapter 10, “Linker Operation”, for more information.

Object modules can be read from a combination of files and are loaded in the order specified, with each subsection within each module being loaded into memory at a higher address than all preceding subsections within its section. Use as many **LOAD** commands as needed. If an absolute path is not provided on a **LOAD** command, the linker searches for a *module* in the following locations and order:

1. Relative path from the current directory
2. **lib** directory parallel to the directory where the linker executable is located. That is, *linker_executable_location*/**../lib**.

Example

```
LOAD a.o,b.o,c.o,d.o,e.o,libc.lib
```

In this example, all input sections from files **a.o**, **b.o**, **c.o**, **d.o**, and **e.o** will be loaded in the order that they are encountered. Sections will be combined based on name and type. The library object modules will not be processed until named object files have been processed and will include only those object modules that define any remaining unresolved externals.

ORDER — Specifies Section Order

Syntax

ORDER ***section***[=*value*][,*section*[=*value*]]...

Description

- **section**

  {*section_name*[![*section_type*][!*object_file*]]
  | !*section_type*[!*object_file*] | *****}

  section_name — Indicates the name of the section.

  section_type — Indicates section type. The only valid types are **TEXT**, **LIT**, **DATA**, and **BSS**.

  object_file — Indicates the input object file name where the section exists.

  ***** — Overrides the first-fit allocation algorithm. There can only be one asterisk, and it must appear last. If an asterisk is present, the [=*value*] clause cannot be used. The first-fit allocation algorithm forces the linker to look for the first available space, starting from address 0, to allocate sections.

- ***value***

  Represents an absolute address expressed as a C constant.

The **ORDER** command specifies the order and/or absolute address of the output sections it names. Specifically, it states the order in which the input sections are to appear in the output file. It overrides the default ordering of output sections, which is **TEXT**, **LIT**, **DATA**, and **BSS**. More than one **ORDER** command may be present, as long as they do not conflict with each other. **ORDER** statements can appear across multiple input lines for clarity.

This command also allows the ordering of specific input sections within a combined output section. An input section is identified by its name and its input object file name. If the name is not unique within an input object module, the section type must be included with the section name. However, the linker will print an informative message concerning any name ambiguities that it encounters. If an input object file is not named with the section name, the section name will be interpreted to mean an output section.

All input sections of the same name and same type are combined into one output section of the same name and type by default.

All sections not specifically located using this directive will be located in the absolute address space on a first-fit basis, according to the default ordering rules. Sections will not span holes in memory; that is, a section will be assigned a starting absolute address and a length. The starting address plus the length then becomes reserved memory and no other section can be assigned within that address space.

To write **ORDER** directives, an understanding of how the linker works is necessary. The linker orders and combines sections after all sections have been read and are known to the linker. The following steps outline the linker algorithm for ordering sections:

1. All memory areas named by the commands **RESNUM** or **RESADD** are reserved.
2. All input sections specified in the **ORDER** command requesting an absolute address are placed at the requested absolute address and are removed from consideration for section combining. They become output sections at this point.
3. All remaining input sections specified in the **ORDER** command are placed in the output section in the order in which they are encountered on the **ORDER** command. All unspecified input sections of the same name and type will be placed after the ordered input sections. This is the same as giving a different **LOAD** input file order but allows individual input sections to be specified rather than all sections of an input file. The important distinction of ordering input sections with the **ORDER** command is that the sections are ordered relative to other input sections of the same type and same name and not how they appear relative to each other on the command statement. Thus, the **ORDER** command can be used to alter the combining rules of the linker.
4. The linker combines sections of the same name and same type into one output section of the same name and type. Alignment of each input section that formed the output section is performed at this phase. The absolute address is assigned based on the final alignment factor, which defaults to the largest alignment value of the input sections being combined into an output section. For linker generated sections, the default alignment is four (4) for text sections and one (1) for data sections.
5. All output sections are now uniquely identifiable and are ready for allocation. The **ORDER** command is scanned for assignment of absolute addresses to output sections. The named output sections are assigned the absolute address.
6. The linker then proceeds to assign addresses to those Small Data Area sections that are not already processed through **ORDER** commands. If the value of the base of the Small Data Area is specified by the user, then the linker tries to relocate the corresponding Small Data Area sections as close to the base values as possible. If the base value is not supplied, then the linker proceeds with relocating Small Data Area sections and it tries to keep the pairs of Small Data Area sections as close together as possible in the available memory. If the special sections **.PPC.EMB.sdata0** and **.PPC.EMB.sbss0** are not processed through **ORDER** commands, then the linker tries to relocate these sections starting from address 0xffff8000 and, if that memory is unavailable, tries again starting at address 0.
7. The linker begins assigning absolute addresses to the remaining output sections, starting at the first nonreserved absolute address available. The **ORDER** command is scanned again looking for placement requests of the output sections. Output sections are placed in the absolute file in the order specified with this command or, if sections are not specifically ordered, by using a first-fit algorithm. That is, the linker looks for an absolute address with sufficient contiguous memory to contain the section it is currently

trying to allocate. Although the linker allocates the output sections in a certain default order (**TEXT**, **LIT**, **DATA**, **BSS**), the first-fit algorithm does not guarantee that the sections will appear in the absolute file in that order. The **ORDER** command must be used to guarantee output order.

8. After all output sections have been assigned their absolute addresses, the linker proceeds to the relocation phase and outputs the final absolute or relocatable object module.

Example

If three input object modules each contribute a section named **.text** of type **TEXT**, the linker will first combine the three sections into one **TEXT** section named **.text**. This section is now an output section that can then be ordered as follows:

```
ORDER .text=value
```

The linker will also accept the following syntax if there is only one output section of type **TEXT**.

```
ORDER !TEXT=value
```

If there are multiple output **TEXT** sections, *value* would be illegal in this context.

Note

In the following examples, assume that input object files are input in the order **a.o**, **b.o**, and **c.o** and consist of the following sections:

a.o	**b.o**	**c.o**
.text!TEXT	text1!TEXT	.text!TEXT
text1!TEXT	.text!TEXT	text1!TEXT
.data!DATA	.data!DATA	.data!DATA
.bss!BSS	.bss1!BSS	.bss!BSS
	.bss!BSS	.rodata!LIT

In the following examples, assume that certain memory regions are reserved by the following linker commands:

```
RESADD 0x1000, 0x1fff
RESADD 0x4000, 0x47ff
RESNUM 0x8000, 100
```

Example 1

```
ORDER .text,!DATA
```

In this example, the linker will combine the input **.text** sections into one output **TEXT** section (**.text!!a.o**, **.text!!b.o**, and **.text!!c.o** are combined to produce **.text!TEXT**). Next, the input **text1** sections (**text1!!a.o**, **text1!!b.o**, and **text1!!c.o** are combined to produce **text1!TEXT**) will be placed into one output section named **text1**. The input **.data** sections will be combined into one **DATA** output section (**.data!!a.o**, **.data!!b.o**, and **.data!!c.o** are combined to produce **.data!DATA**). The input **.bss** sections will be combined into one output **BSS** section (**.bss!!a.o**, **.bss!!b.o,** and **.bss!!c.o** are combined to produce **.bss!BSS**).

After all the relocatable sections have been combined and the output sections are known, the linker will allocate absolute addresses to them. The linker will look for the first address available that has a contiguous address space of the size of the section it is trying to locate. So, if the size of the output section **.text** is smaller than 0x1000, it will be assigned the absolute address of 0. If its size is larger than 0x1000, the linker will look for the next largest "hole" available in which the **TEXT** section can be placed.

After the **.text** section is allocated, the linker will then allocate all sections whose section type is **DATA**. The linker will write any remaining unordered sections using the default ordering rules (**TEXT** sections, **LIT** sections, **DATA** sections, and **BSS** sections).

Note that this does not mean that output sections will be placed in the absolute address space in a specific order; rather, it simply means that they will be viewed for placement by the linker in a specific order. The first-fit algorithm will be used to place output sections in the absolute file unless this directive specifically asks for placement order.

In this example, the **.text** section will be allocated the first available absolute address and then **.data** will be allocated the next following available address. The linker now reverts back to the default, which is to assign the first absolute address available, not the following available address, to the **text1** section. Then the **.rodata** section will obtain the first available absolute address that it can fit in. The linker continues in this manner for the **.bss** section and the **bss1** section.

Example 2

```
ORDER .text,!DATA,*
```

In this example, the first-fit algorithm of the linker is overridden. This shows a short-hand method of ordering output sections. The linker proceeds exactly as in the previous example above, except that it never reverts back to the default allocation. In this example, the output **.text** section will be allocated first, then all output sections of type **DATA** will be allocated after the **.text** section, then remaining **TEXT** sections will be allocated after the **DATA** sections, followed by the **LIT** sections, and finally by the **BSS** sections.

Example 3

```
ORDER .text!!a.o,.text!!c.o,.text!!b.o,.text=0x5000
```

This example demonstrates how one output section is to be combined. In the default case, an output section results from combining input sections of the same name and type in the order in which they were input. This command overrides that ordering by specifying that the **.text** output section be arranged as if the input sections came from **a.o**, **c.o**, and **b.o**, respectively. Once combining has finished, the linker sees that absolute addresses are requested for an output section, **.text** in this example. The linker will now assign the first available absolute address to **.text**. Then the linker will follow the default rules of allocating the remaining output sections, which is to locate any remaining **TEXT** sections, **text1** in this example, then allocate any **LIT** sections, then **DATA** sections, and finally, the **BSS** sections.

Example 4

```
ORDER .text=0x1000
```

In this example, an error would be produced but would not be reported until absolute addresses were assigned to the output sections. That is, the linker would combine all of the sections and then allocate absolute output section requests, only to discover that 0x1000 was already reserved. A final output module would not be produced in this instance, and an informative error message would be printed stating the reason for this action.

Example 5

```
ORDER .text!!a.o=0x10000
```

In this example, **.text!!a.o** will be treated as if this section was input as an absolute section. It will not participate in combining. It will be assigned the absolute address of 0x10000. Then the linker will combine the remaining **TEXT** input sections in the order in which they are input to the linker, creating one output **.text** section. The sections are combined in the following order: **text1**, **LIT**, **DATA**, and **BSS.** The linker will then allocate unreserved absolute addresses to **.text**, **text1**, **.rodata**, **.data**, **.bss**, and **bss1** output sections, perform the required relocation, and output the final absolute output module.

Example 6

```
ORDER .bss!!b.o=0,.bss=0x10000
ORDER .data!!b.o=0x20000,.data=0x2000
```

In this example, **.bss!!b.o** will be made into an absolute section and assigned the absolute address of 0. This section will not appear with any other combined **.bss** section in the output file. Then **.data!!b.o** will be made absolute and assigned the absolute address of 0x20000. This section will not appear with any other combined **.data** section in the output file. Since there are no more input sections to be made absolute, the linker will go through normal combining, as in the first example, and begin allocating absolute addresses to the output sections. Since this command specifies addresses for output sections, the linker will assign the absolute address of 0x2000 to the output **.data** section. The combined output **.bss** section will be located at 0x10000. All remaining sections will be assigned addresses according to the default ordering rules and memory availability. Thus, the **.text** output section will be placed in the first available memory area that can contain it. Then **text1** will be allocated, followed by the **.rodata** section, and finally by the **bss1** section.

Example 7

```
ORDER .text,.data,.rodata,bss1,.bss,*
```

In this example, all output sections will be formed and ordered as requested. The output section, **text1**, will be placed after **.bss** since **text1** was not specified with this command.

PUBLIC — Reassigns Global Symbol Value

Syntax

PUBLIC ***newsymbol*** = { *value* | *oldsymbol* [{+|-} *value*] }

Description

- **newsymbol**

  Specifies a global symbol to be defined.

- ***oldsymbol***

  Specifies a global symbol whose value is to be assigned to the new global symbol.

- ***value***

  Specifies a value to be assigned to the new global symbol. *value* is a constant. If a value is specified in addition to an existing global symbol, the value is added to or subtracted from the address or value of the global symbol.

The **PUBLIC** command defines or changes the value of a global symbol. Symbol names specified by the linker's **PUBLIC** command take precedence over symbol names defined during assembly. If a symbol specified by this command is already a global symbol, the value of the symbol is changed to that specified in the **PUBLIC** command.

If the symbol is not already defined, the linker will enter it into the symbol table along with the specified value. The symbol will then be available to satisfy external references from object modules. Any constant value defined by this command is considered to be absolute; it does not lie in any relocatable section and will not be relocated relative to any section base. If a relocatable symbol is specified, the public symbol will be relocated relative to the same section.

Notes

This command lets you specify the value of some external symbols at link time, which might help developers avoid reassembly. The new value can be specified either before or after the object module containing the symbol is loaded. Note that defining this symbol before a library containing that symbol is loaded will mean the module containing that symbol may not be loaded, depending upon whether any other public symbols within that module resolve any unresolved externals. If the symbol is defined in the linker command file after the library containing the public symbol is loaded, then the module containing that symbol will be loaded, but the value of the symbol will be changed to the value specified in the linker command file.

Example

```
PUBLIC INPUT = 0
PUBLIC NEWADDR = OLDADDR + 8
```

The value of the symbol **INPUT** is defined to be 0.The value of the symbol **NEWADDR** is defined to be 8 greater than the value of **OLDADDR**, whether **OLDADDR** is a relocatable value or a constant value.

RESADD/RESNUM — Reserves Memory

Syntax

RESADD ***low_addr***,***high_addr***

RESNUM ***low_addr***,***number***

Description

- **low_addr**

  The lowest address to be declared off-limits.

- **high_addr**

  The highest address to be declared off-limits.
  high_addr >= *low_addr*.

- **number**

  The number of bytes including *low_addr* to be declared off-limits. If *number* is 0, no area is reserved.

The **RESADD** and **RESNUM** commands declare certain areas of memory as off-limits to the linker. No relocatable code is placed in these areas. You might wish to use these commands to avoid overwriting an operating system in low memory, for example.

RESADD reserves the addresses *low* to *high*, inclusive. **RESNUM** reserves the addresses *low* to *low*+(*number*-1).

```
Examples
RESADD 0x8000,0x9FFF

RESNUM 0x8000,0x2000
```

In this example, both commands reserve 8192 (2000 hexadecimal) bytes of memory, so nothing will be loaded in this area. This reserved region is essentially a “hole” in memory.

SECTSIZE — Sets Maximum Section Size

Syntax

SECTSIZE ***section*** = ***value*** [, *section* = *value*...]

Description

- **section**

  {*section_name*[!*section_type*]}

section_name	Indicates the name of the output section
section_type	Indicates section type. The only valid types are **TEXT**, **LIT**, **DATA**, and **BSS**.

- **value**

  Constant value indicating the maximum size for the indicated section.

The **SECTSIZE** command is used to make sure that a section is of a given size. If the total section size for an indicated section is less than the value used with this command, the section's size will be increased to match that value and an informational message is emitted to indicate this change. If the total section size for an indicated section is larger than the value used with this command, a warning is emitted to indicate that the command had no effect.

The **SECTSIZE** command is especially useful for setting the size of **BSS** sections, such as for stacks or heaps. If the indicated section does not exist, a section will be created by the linker. The section will have the type **DATA** and will be initialized with 0s.

Example

```
SECTSIZE .bss = 0x2000
```

If the total size of all the **.bss** module sections is less than 0x2000 bytes (including padding), then the size of the **.bss** section will be set to 0x2000. If the total size is greater than 0x2000 bytes, the linker will emit a warning that the command could not be followed.

START — Specifies Entry Point

Syntax

START {*symbol* | *value*}

Description

- ***symbol***

  Represents a global symbol.

- ***value***

  Specifies a location in memory. *value* is a constant.

The **START** command specifies the point at which program execution is to begin. This command is used to specify a specific entry point or to override the previously defined or default entry point. If neither the **START** command nor the **-e** linker command line option is specified, the linker will look for the symbol **__mri_start** as the entry point. If **__mri_start** does not exist, the entry point will be set to zero, and a warning message will be produced. Any specified symbol or value must be evenly divisible by four (a 4-byte address boundary).

Example

```
START entry
```

In this example, the entry point is set to the address of the symbol **entry**.

SYMTRAN — Modifies Global Symbol Names

Syntax

SYMTRAN [/LEADING] ***matchstring*** [*replacestring*]

SYMTRAN /EMBEDDED ***matchstring*** [*replacestring*]

SYMTRAN /TRUNCATE ***value***

Description

- **LEADING**

  Modifies the patterns that are matched at the beginning of a global symbol. If the first string is found to match the text at the beginning of a global symbol, the matching text is removed and is replaced with the replacement string, or nothing if no replacement string is indicated. It is acceptable to use an empty string, as indicated by two quote characters, for the matching string. This could, in effect, be used to insert a string at the beginning of all global symbols.
  (default type of matching)

- **matchstring**

  A pattern to be searched for in a global symbol. This pattern is always removed from the symbol. The pattern may or may not be quoted. If spaces or tabs are part of the pattern, then quotes must be used.

- ***replacestring***

  A pattern to be used to substitute for the matched pattern. The pattern may or may not be quoted. If spaces or tabs are part of the pattern, then quotes must be used.

- **EMBEDDED**

  Modifies patterns that exist anywhere within a global symbol. If the first string is found to match the text within a global symbol, the matching text is removed and is replaced with the replacement string, or nothing if no replacement string is indicated. The string to be matched must contain at least one character.

- **TRUNCATE**

  Sets a maximum symbol length for global symbols. Any value from 1 to 8192 is valid. This maximum symbol length is used from the time this command is used until either the end of the command file or another **/TRUNCATE** size is set.

- **value**

  An absolute constant that represents the maximum length for all emitted global symbols.

The **SYMTRAN** command is used to modify global symbols from one form into another. It has three different types of operations:

- **LEADING**
- **EMBEDDED**

- **TRUNCATE**

The command modifiers (that is, the option specified after the '/') must be spelled out either completely or with enough characters to differentiate the operation from the others. For now, even one character is enough to differentiate between the three valid operations.

If the result of a replacement causes the resulting symbol to exceed the maximum symbol length of the linker, the symbol will be truncated to the maximum symbol length and all searching stops. This maximum symbol length is not the length as specified by the **/TRUNCATE** command modifier.

If multiple **SYMTRAN** commands of the same type of operation (**/LEADING** or **/EMBEDDED**) are specified, the last command defined will be checked for first and the rest will be searched in reverse order of definition. The only variation on this behavior is when the same matching pattern is specified more than once. In that case, the replacement text of the earlier use of the matching string is replaced and no new checks are added.

When multiple matching patterns exist for the same type of operation, the linker will make a pass through all of the matching patterns for a given location in the global symbol. For the **/LEADING** operation, only the first location is checked. For the **/EMBEDDED** operation, the linker will make a pass through the patterns for each character in the symbol. If no matches occur for a given location, the linker moves to the next character. If there is a match, the linker will perform each possible replacement for that position, going once through the entire list of possible matching patterns. The linker will then skip over any characters in the last replacement text and begin searching at that location. The searching will continue until the end of the symbol is reached.

The three operations are performed on global symbols from the position of the command until the end of the command file. The order of processing is to perform all **/LEADING** operations first, followed by all **/EMBEDDED** operations, followed by the truncation of the symbol.

This command can be used for both absolute and incremental linking.

Example

```
SYMTRAN /EMBEDDED "_"
PUBLIC _SYM_1_; _SYM_1_ -> SYM1

SYMTRAN /EMBEDDED "ABA" "X"
PUBLIC ABABABABABA; ABABABABABA -> XBXBX

SYMTRAN "ABA" "X"
PUBLIC ABABABABABA; ABABABABABA -> XBABABABA

SYMTRAN /LEADING "_"
PUBLIC _SYM_1_; _SYM_1_ -> SYM_1_

SYMTRAN . _
PUBLIC .SYM_2_; .SYM_2_ -> _SYM_2_

SYMTRAN /TRUNCATE 4
PUBLIC _SYM_1_; _SYM_1_ -> _SYM
```

```
SYMTRAN /LEADING "" "_"
PUBLIC SYM; SYM -> _SYM
```

Chapter 12
Sample Linker Session

The LNKPPC Linker uses a two-pass process. During the first pass, the linker checks the commands and object modules for errors and constructs a symbol table. During the second pass, it produces the output object module and, if requested, a listing.

The final absolute object module is produced, along with a link map, during the second pass of processing. You can set command line flags to list a global symbol table and/or a cross-reference table in the link map. The link map also indicates the starting address of the absolute file, the output module name and format, and the section and module summary.

If the inclusion of a module from a library adds an undefined reference, the linker searches the library repeatedly until all possible external references have been resolved. If unresolved externals still exist after all object modules and libraries have been processed, an absolute object module is still generated, but an error is produced.

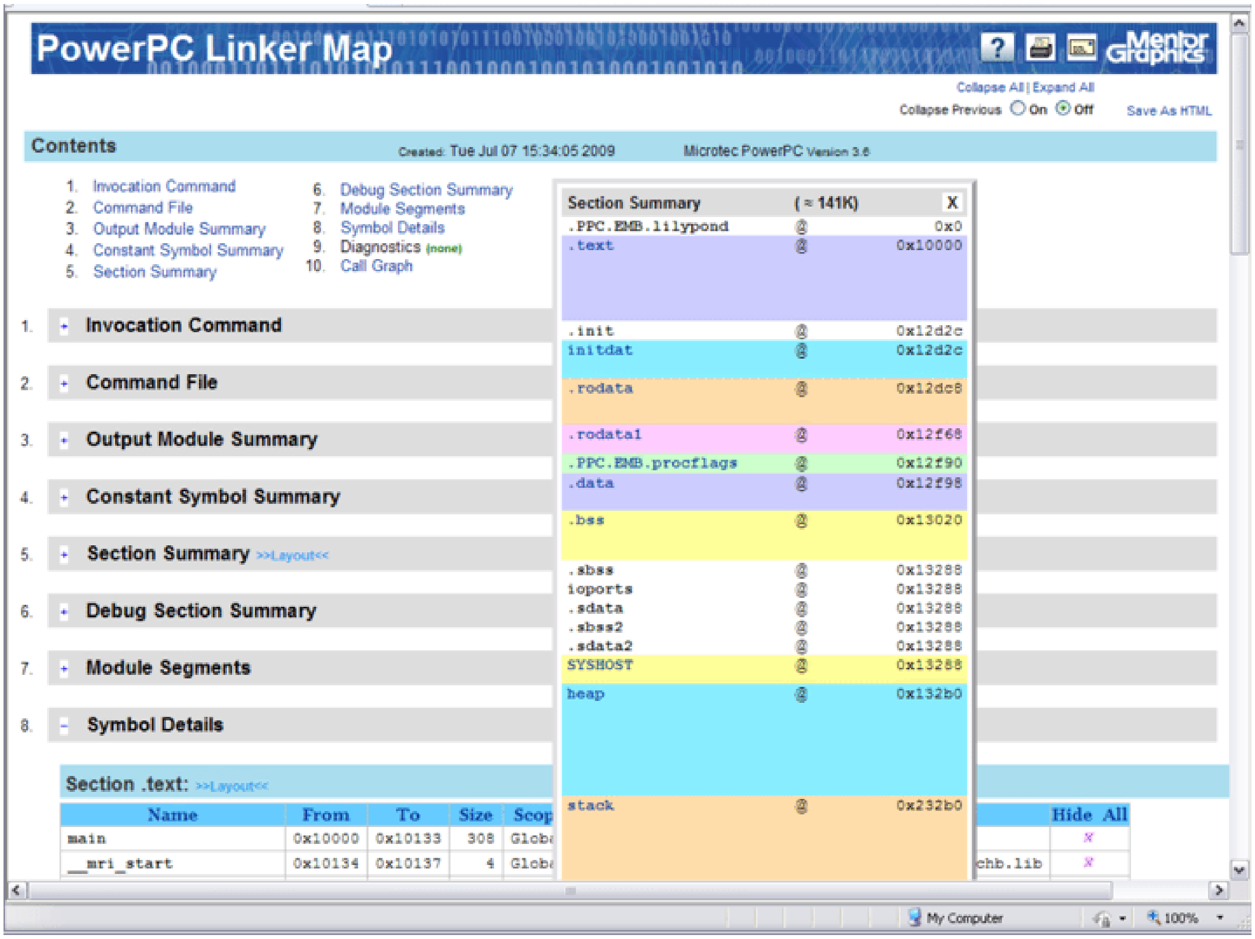
The main purpose of an output linker listing is to convey all pertinent information regarding the linking process. The listing can also be used as a documentation tool by including comments and remarks that describe the function of the particular program section.

LNKPPC is capable of producing two types of linker maps, an XML version and a text version. The type of linker map generated is dependent on the extension supplied with the map file name.

XML Linker Map

The map produced by the linker (option **–m**, or *–mfilename.xml*) is an example of the graphics-based modern output which is in XML. The map is interactive and should be viewed with a Web browser. Its extension should be *.xml* or your browser may not recognize it. The best approach is to try "mccppc –m hello.c" and then bring up *hello.xml* in your browser. You can explore the features such as expand/collapse, sort by any key, click on column headers, hide, display the call graph of a symbol, and so on.

Figure 12-1. XML Linker Map



PowerPC Linker Map

Collapse All | Expand All
Collapse Previous ○ On ● Off   Save As HTML

Contents   Created: Tue Jul 07 15:34:05 2009   Microtec PowerPC Version 3.6

1. Invocation Command
2. Command File
3. Output Module Summary
4. Constant Symbol Summary
5. Section Summary
6. Debug Section Summary
7. Module Segments
8. Symbol Details
9. Diagnostics (none)
10. Call Graph

1. + Invocation Command

2. + Command File

3. + Output Module Summary

4. + Constant Symbol Summary

5. + Section Summary >>Layout<<

6. + Debug Section Summary

7. + Module Segments

8. - Symbol Details

Section Summary	(≈ 141K)	X
.PPC.EMB.lilypond	@	0x0
.text	@	0x10000
.init	@	0x12d2c
initdat	@	0x12d2c
.rodata	@	0x12dc8
.rodata1	@	0x12f68
.PPC.EMB.procflags	@	0x12f90
.data	@	0x12f98
.bss	@	0x13020
.sbss	@	0x13288
ioports	@	0x13288
.sdata	@	0x13288
.sbss2	@	0x13288
.sdata2	@	0x13288
SYSHOST	@	0x13288
heap	@	0x132b0
stack	@	0x232b0

Section .text: >>Layout<<

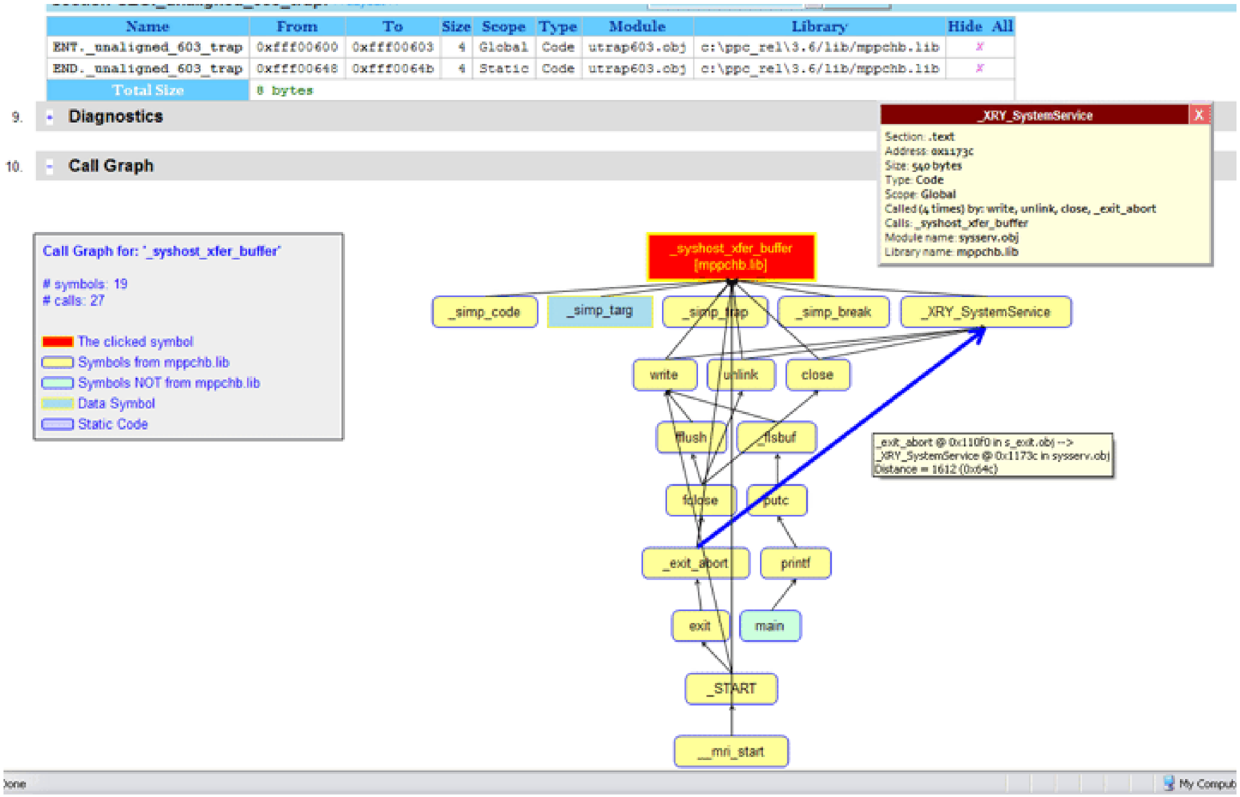
Name	From	To	Size	Scop		Hide All
main	0x10000	0x10133	308	Glob		☒
__mri_start	0x10134	0x10137	4	Glob	chb.lib	☒

My Computer   100%

To view the help, click the ? icon in the top right corner of the map file. This opens *help.html* which has the same look and feel as the map file and highlights the provided features.

The linker provides an interactive call graph feature you can explore while browsing a linker map. Figure 12-2 is an example of a call graph.

Figure 12-2. Call Graph Example



Name	From	To	Size	Scope	Type	Module	Library	Hide All
ENT._unaligned_603_trap	0xfff00600	0xfff00603	4	Global	Code	utrap603.obj	c:\ppc_rel\3.6/lib/mppchb.lib	✗
END._unaligned_603_trap	0xfff00648	0xfff0064b	4	Static	Code	utrap603.obj	c:\ppc_rel\3.6/lib/mppchb.lib	✗
Total Size	8 bytes							

9. Diagnostics

10. Call Graph

Call Graph for: '_syshost_xfer_buffer'

symbols: 19
calls: 27

🟥 The clicked symbol
▭ Symbols from mppchb.lib
▭ Symbols NOT from mppchb.lib
▭ Data Symbol
▭ Static Code

_XRY_SystemService

Section: .text
Address: 0x1173c
Size: 540 bytes
Type: Code
Scope: Global
Called (4 times) by: write, unlink, close, _exit_abort
Calls: _syshost_xfer_buffer
Module name: sysserv.obj
Library name: mppchb.lib

_exit_abort @ 0x110f0 in s_exit.obj -->
_XRY_SystemService @ 0x1173c in sysserv.obj
Distance = 1612 (0x64c)

Text-Based Linker Map File Listing

LNKPPC is also capable of producing a text-based linker map. This can be obtained by specifying the extension “map” after the file name. For example, specifying the option “-mfilename.map” would produce a text-based linker map called *filename.map*.

The following pages describe the text-based listing format. Figure 12-3 shows a sample output listing. Refer to the following numbered points in the figure in order to examine the listing.

1. The page headings show the name of the Microtec tool, its version number, and the time and date of the program run.
2. The line entitled **Command line:** specifies the command line used to invoke the linker.
3. Commands are listed from the command file. A description of the commands shown in the map file is as follows:
   - The **RESADD** command causes the linker to reserve space in memory. In this case, addresses from 0x100 to 0x300 are reserved.
   - The **ORDER** command arranges output sections in the final absolute object module. In this case, section **.text** is placed at 0x1000, section **.rodata1** is placed at 0x2000. These sections are followed by sections of type **DATA** and section **.bss**.

- The **ALIGN** command causes the named text section, **.text**, to begin on 4-byte boundaries in memory.
- All other sections not specified by the **ORDER** command are placed at an absolute address on a first-fit basis, starting from address zero.
- The **START** command specifies the value to be used for the starting address. In this case, the start address is the value of the public symbol **main**.
- The **LOAD** command specifies the order of input files. The linker orders output sections based on the order in which sections are input. In this directive, all sections from the object file **sample1.o** will be read first.
- Additional input files are input with multiple **LOAD** directives. In this case, all sections of the object file **sample2.o** will be input. The end of the load directives is detected by an **end-of-file** condition, and the linker proceeds to allocate the sections as requested and output the final object module.

Module names provided on the command line are translated into load commands in the order that they appear on the command line. They are inserted prior to any **LOAD** directives in the command file and are displayed along with other commands encountered in the command file.

Errors detected during the processing of a command file such as syntax errors, conflicts of reserved memory, bad or missing load modules, and unresolved externals are displayed in this section.

4. The lines entitled **OUTPUT MODULE NAME:** and **OUTPUT MODULE FORMAT:** list the output module name and whether the object module is absolute or relocatable. Section order violations are listed in this area to better offset them from the section summary.

   The **TARGET PROCESSOR:** line lists the processor type that the output file is expected to run on. If none of the input objects indicated a processor type, the line indicates UNKNOWN. If the output object file is usable for all PowerPC microprocessors, the line will indicate **COMPATIBLE**. Otherwise, the specific processor type is indicated. Some additional information that is noted after the processor type is as follows:

   - (COMPATIBLE) The indicated processor type is not a requirement. The object file does not contain any instructions that are target specific. If this information is not displayed, the output object should only be used on the indicated processor type.
   - (ILLEGAL CONTENTS) One of the input object files contains instructions that are not valid for the indicated processor type. Care should be taken in using the output object file.
   - (INCOMPLETE INFO) The Linker was not able to find processor information in one or more of the input object files. Therefore, it is possible that the indicated

processor type or that the contents of the output object file are not correct. Care should be taken in using the output object file.

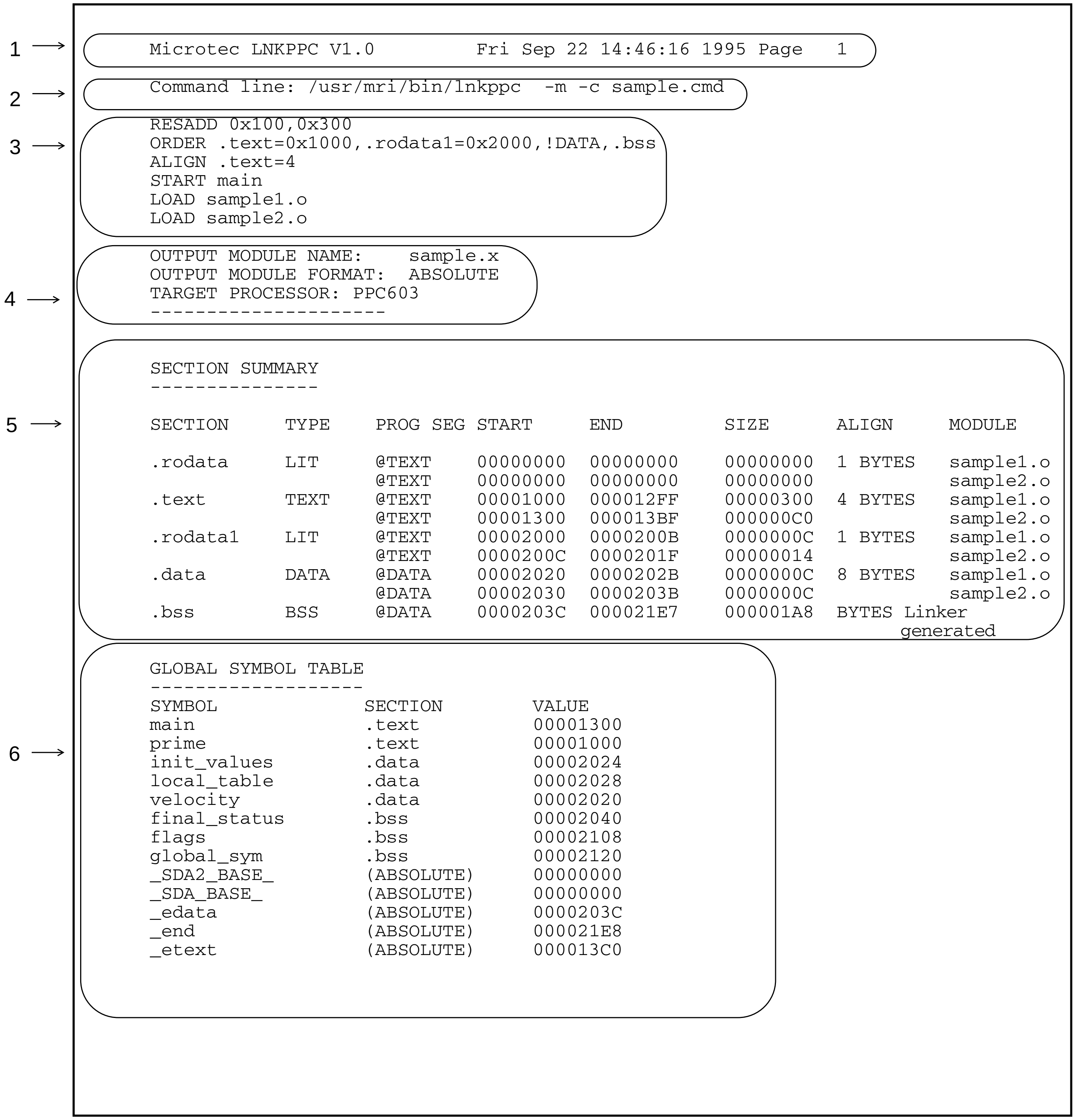
- (SIMULATOR REQUIRED) The output object file contains instructions that are not executed in hardware on the indicated processor type. Software simulation of these instructions is required in order to guarantee that the output object file will execute appropriately. The absence of this message indicates either the target processor is able to execute any of the instructions in the output object file or that a software simulator has been linked in to handle those instructions.

5. The line entitled **SECTION SUMMARY** lists the sections that have been linked. The summary is sorted by address and gives each section's attribute, length, and alignment.
6. The area entitled **GLOBAL SYMBOL TABLE** lists those symbols that have been declared global in the assembler. The symbol's name, section in which it is defined, and virtual address are displayed. These symbols are sorted by name and type. This table is listed by default; you can suppress it by using command line options.
7. The **CROSS REFERENCE TABLE** is produced by default. The area entitled **CROSS REFERENCE TABLE** shows a cross-reference of modules that define and refer to public symbols. Module names with a minus sign (**-**) indicate that the symbol was defined in this module. Module names without the minus sign denote a module containing a reference to the symbol.
8. The **START ADDRESS** is the address to which the initial program counter will be assigned. It is obtained from the **-e** option or the **START** command. If neither of these are specified, the entry point address is set to zero, and a message stating that fact appears.

All values, such as the start, end, and length of the section summary and the value of the symbol table, are given in hexadecimal.

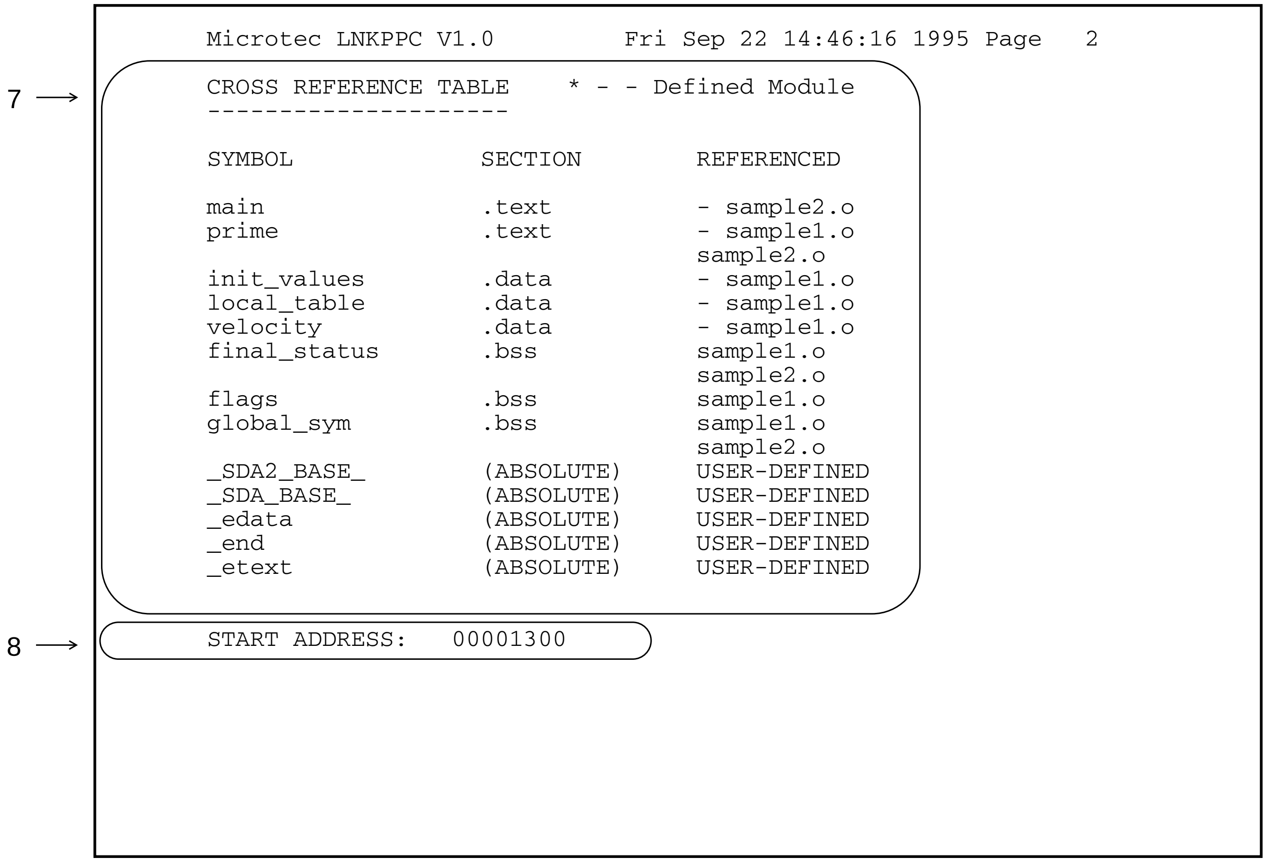
Sample Linker Map File

Figure 12-3. Sample Link Map Listing



```
Microtec LNKPPC V1.0         Fri Sep 22 14:46:16 1995 Page   1

Command line: /usr/mri/bin/lnkppc  -m -c sample.cmd

RESADD 0x100,0x300
ORDER .text=0x1000,.rodata1=0x2000,!DATA,.bss
ALIGN .text=4
START main
LOAD sample1.o
LOAD sample2.o

OUTPUT MODULE NAME:    sample.x
OUTPUT MODULE FORMAT:  ABSOLUTE
TARGET PROCESSOR: PPC603
---------------------


SECTION SUMMARY
---------------

SECTION     TYPE    PROG SEG START     END         SIZE       ALIGN     MODULE

.rodata     LIT     @TEXT    00000000  00000000    00000000   1 BYTES   sample1.o
                    @TEXT    00000000  00000000    00000000             sample2.o
.text       TEXT    @TEXT    00001000  000012FF    00000300   4 BYTES   sample1.o
                    @TEXT    00001300  000013BF    000000C0             sample2.o
.rodata1    LIT     @TEXT    00002000  0000200B    0000000C   1 BYTES   sample1.o
                    @TEXT    0000200C  0000201F    00000014             sample2.o
.data       DATA    @DATA    00002020  0000202B    0000000C   8 BYTES   sample1.o
                    @DATA    00002030  0000203B    0000000C             sample2.o
.bss        BSS     @DATA    0000203C  000021E7    000001A8   BYTES Linker
                                                                   generated

GLOBAL SYMBOL TABLE
-------------------
SYMBOL             SECTION        VALUE
main               .text          00001300
prime              .text          00001000
init_values        .data          00002024
local_table        .data          00002028
velocity           .data          00002020
final_status       .bss           00002040
flags              .bss           00002108
global_sym         .bss           00002120
_SDA2_BASE_        (ABSOLUTE)     00000000
_SDA_BASE_         (ABSOLUTE)     00000000
_edata             (ABSOLUTE)     0000203C
_end               (ABSOLUTE)     000021E8
_etext             (ABSOLUTE)     000013C0
```

Figure 12-4. Sample Link Map Listing (cont.)



```
Microtec LNKPPC V1.0        Fri Sep 22 14:46:16 1995 Page   2

CROSS REFERENCE TABLE    * - - Defined Module
---------------------

SYMBOL               SECTION        REFERENCED

main                 .text          - sample2.o
prime                .text          - sample1.o
                                    sample2.o
init_values          .data          - sample1.o
local_table          .data          - sample1.o
velocity             .data          - sample1.o
final_status         .bss           sample1.o
                                    sample2.o
flags                .bss           sample1.o
global_sym           .bss           sample1.o
                                    sample2.o
_SDA2_BASE_          (ABSOLUTE)     USER-DEFINED
_SDA_BASE_           (ABSOLUTE)     USER-DEFINED
_edata               (ABSOLUTE)     USER-DEFINED
_end                 (ABSOLUTE)     USER-DEFINED
_etext               (ABSOLUTE)     USER-DEFINED

START ADDRESS:   00001300
```

Chapter 13
Librarian Operation

The LIBPPC Object Module Librarian builds program libraries. Libraries are collections of relocatable object modules residing in a single file. These libraries allow the linker to automatically load frequently used object modules that define global symbols referenced in other loaded modules. These modules are linked without concern for the specific names and characteristics of the modules in which the symbols are defined.

Librarian Features

Features of the LIBPPC Object Module Librarian include:

- Ability to combine multiple Microtec IEEE-695 or SYS V PORTAR libraries into a single library
- Ability to include modules from other Microtec IEEE-695 or SYS V PORTAR libraries
- Ability to convert Microtec IEEE-695 format libraries to the SYS V PORTAR library format, or to convert SYS V PORTAR format libraries to the Microtec IEEE-695 library format.

The LIBPPC Librarian supports both the Microtec IEEE-695 library format and the System V (SYS V)PORTAR library format. The default behavior of the librarian is to create Microtec IEEE-695 libraries, but the SYS V PORTAR library format can be selected through either a command line option or the FORMAT command. Once a library is created in one of the two formats, it continues to remain in that format unless the output format is changed through the command line option or the FORMAT command. The contents of either library format continues to be ELF relocatable objects.

This chapter describes how to build and modify the libraries and how the linker uses the libraries. When used in connection with the librarian, the word “module” refers to a relocatable object module that results from assembling a source program with the ASMPPC Assembler, or from linking two or more relocatable object modules incrementally. Error messages and warnings are listed in Appendix D, “Librarian Error Messages”.

Librarian Function

The librarian formats and organizes library files for use by the linker. Libraries provide a convenient means for managing collections of relocatable object modules. Through the use of libraries, linkers can easily access relocatable object modules when required. This efficiency comes from reducing the number of files that must be opened in order to link modules.

When writing modular programs, communication among the various modules is established through the use of global and external symbols. For example, Figure 13-1 shows three relocatable object modules that result from an assembly.

Figure 13-1. Three Relocatable Object Modules Resulting From Assembly

```
.file Module_1.s

   .extern fallow
   .globl next
   ba fallow

next:
   nop
.end
```

A relocatable object module that resides in host system file **KNEWEL.o**

```
.file Module_2.s

   .extern next
   .globl fallow

fallow:
   add r3,r4,r4
   nop
addis  r5,r0,lo(next)
.end
```

A relocatable object module that resides in host system file **SWIGGET.o**

```
.file Module_3.s

   .globl arctan

arctan:
   or r3,r4,r3
.end
```

A relocatable object module that resides in host system file **BAYER.o**

Of the three modules shown, **Module_1.s** and **Module_2.s** communicate with one another through external references and global symbols, while **Module_3.s** is a stand-alone module.

The relocatable modules illustrated consist of load data information, relocation information, and records that indicate global symbols and external symbols.

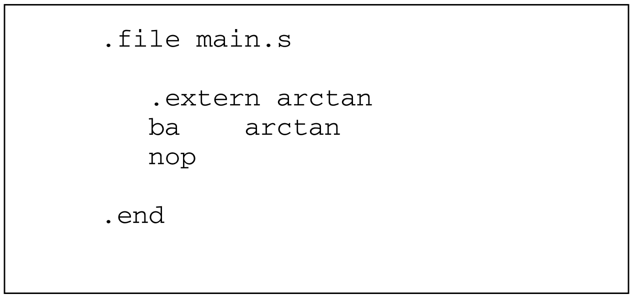
By using various combinations of librarian commands, the relocatable object modules shown can be made members of a library. For example, a new library can be created by using the following commands:

```
CREATE NEWREM.LIB
ADDMOD KNEWEL.o
ADDMOD SWIGGET.o,BAYER.o
SAVE
```

Now the library can be used by the linker.

Assume that you have written a program module called **MAIN**. After **MAIN** has been assembled, the resultant relocatable object module in Figure 13-2 is in a host system file named **MAIN.o**. This module has a reference to the global symbol **ARCTAN**.

Figure 13-2. Relocatable Object Module in MAIN.o



```
.file main.s

   .extern arctan
   ba    arctan
   nop

.end
```

A relocatable object module that resides in host system file **MAIN.o**

Before the library existed, you could have directed the linker to load the **MAIN** module and the module that contains the reference to **arctan** as follows:

```
LOAD MAIN.o
LOAD BAYER.o
```

After the library has been created, you can direct the linker as follows:

```
LOAD MAIN.o
LOAD NEWREM.LIB
```

The linker will access the library to try to resolve external references such as **ARCTAN**. The **MAIN** module can be modified so it calls the **fallow** subroutine as well (see Figure 13-3).

Figure 13-3. MAIN Module Modified to Call KNEWEL.o Module

```
   .file main.s

      .extern  arctan
      .extern  fallow

      ba       arctan
      ba       fallow
   .end
```

A relocatable object module that resides in host system file **MAIN.o**

Without the ability to link from a library, it would be necessary to command the linker as follows:

```
LOAD MAIN.o
LOAD SWIGGET.o
LOAD BAYER.o
LOAD KWEWEL.o ; Referenced by SWIGGET.o
```

When using a linker that has the ability to load from a library, you need only to specify:

```
LOAD MAIN
LOAD NEWREM.LIB
```

The linker will load the relocatable object module **MAIN** in the usual way. It will load the other modules from the library referenced by **MAIN**.

The following example is a more practical illustration of using the library.

Example 13-1. Using the Library

Suppose you write a series of program modules consisting of a number of mathematical routines including a few modules that calculate transcendental functions. These modules are then gathered into a library file by the LIBPPC Object Module Librarian.

Sometime later, you need to calculate an arc tangent function within a program being written. You are aware of the fact that there is an arc tangent function in a library file and you know the name of the entry point of the routine. You also know how to pass parameters to the arc tangent function and how to accept the result of the calculation.

You need only do the following:

1. Call the arc tangent function from the program being developed, placing the global name of the entry point into the argument field of the **BA** instruction.

2. Place the global entry point name of the arc tangent function in the argument field of an external reference pseudo-operation in the program being written (using the assembler **.GLOBL** or **.EXTERN** directives).

Even without the name of the relocatable object module that contains the arc tangent function, you can include the correct relocatable object module by informing the linker to use the library file containing the collection of transcendental functions.

You do not have to specify which module contains the arc tangent function. The linker automatically searches the named library. It looks for the entry point name coded as the argument of the calling statement. When the entry point name has been found, the linker identifies the module in which it resides and then includes that module in the current load.

The linker determines which of the library modules to use by examining the internal list of unresolved external references accumulated during the link process. It then accesses the library file to determine if there is a match between unresolved external references and a label or name that has been declared global in the library file modules. The linker then identifies which modules contain the matching global symbols and includes those modules just as if you had explicitly directed the linker to load the proper modules.

When the inclusion of a module in the library adds an undefined reference to the list of undefined references, the linker will access the library again until all external references have been satisfied. All global symbols within a library must have unique names.

Return Codes

The librarian provides operating system-specific return codes. The librarian either completes without encountering an error, displays a message or warning, or terminates with an error. Error messages and warnings are listed in Appendix D, “Librarian Error Messages”. Return codes are described in the section Return Codes in Chapter 2, Command Usage.

Chapter 14
Librarian Commands

This chapter describes the commands used by the LIBPPC Object Module Librarian. The librarian reads a sequence of commands from the command input device in interactive or batch mode. The command sequence must be terminated by the **END** command. Relocatable object modules are read as input and collected in organized libraries as specified in the command input file.

Multiple sessions are permitted through the use of commands. A session is a list of commands before a **CLEAR** command. The session does not end until this command is given. The **END** command terminates the current session and exits the librarian.

Command Syntax

Librarian module names are the same as the object file name (without the path). Each module must have a unique name. Module names are case-sensitive. Public symbols are written according to the same rules as in the assembler.

The librarian recognizes several special characters. These characters, and the functions they perform when used in a library command line, shown in Table 14-1.

Table 14-1. Librarian Special Characters

Character	Command Line Function
***** — asterisk **;** — semicolon	Places a comment in a command sequence. The librarian ignores the rest of the line following these characters. The librarian does not process comments; it writes them to the standard list device, usually the terminal.
(— left parenthesis **)** — right parenthesis	Denote a list of similar elements in a command. Parentheses are used in pairs. They can only be used to group module names that are members of a library.
, — comma	Separates members of a list of similar elements. The list can contain module names or module file names.

Table 14-1. Librarian Special Characters

Character	Command Line Function
+ — plus sign	Acts as a continuation character. When followed by a carriage return, the plus sign allows you to continue a list on one or more subsequent lines. Exercise care when using line continuation: do not break up or interrupt a complete syntactical unit such as a filename, a module name, or a command. The command verb must be terminated by a blank if it is an argument. If the continuation character is used immediately after the command verb, it must be separated from the command by at least one blank. Except as noted here, the line continuation character can appear anywhere in a command.

spaces
tab characters
	Can be used freely within commands. Whitespace (spaces and tab characters) can only be used between syntactically identifiable units. It is not valid to put blanks inside commands, module names, and so forth. Whitespace is ignored except in the function of acting as a separator. For example:
	`DELETE mod1.o, mod2.o`
	is the same as:
	`DELETE mod1.o,mod2.o`

Command File Comments

Comments can be included in a command file to document the processing. These comments are included by use of the semicolon (**;**) or asterisk (*****):

```
; this is a complete line of comment
  addmod modulea.o  ; this is a command line comment
  addmod moduleb      * this is another comment
```

Command Summary

Table 14-2 lists the librarian commands and abbreviations. Commands are detailed in the following pages. Abbreviations indicate the minimum set of characters needed to recognize a command.

Table 14-2. Librarian Commands and Abbreviations

Command	Abbreviation	Description
ADDLIB	ADDL	Adds module(s) from another library
ADDMOD	ADDM	Adds object module(s) to current library
CLEAR	CL	Clears library session since last **SAVE**
CREATE	CR	Defines new library
DELETE	DE	Deletes module(s) from current library

Table 14-2. Librarian Commands and Abbreviations (cont.)

Command	Abbreviation	Description
DIRECTORY	DI	Lists library modules
END/QUIT	EN/Q	Terminates librarian execution
EXTRACT	EXT	Copies library module to a file
EXTRACTALL	EXTRACTA	Extracts all modules from current library
FORMAT	FO	Sets output library type
FULLDIR	FU	Displays library or library module contents
HELP	H	Displays context-sensitive command syntax
OPEN	O	Opens an existing library
REPLACE	R	Replaces library module
SAVE	S	Saves contents of current library

ADDLIB — Adds Module(s) From Another Library

Abbreviation

ADDL

Syntax

ADDLIB ***library_filename***[(*module_name*[,*module_name*]...)]

Description

- **library_filename**

  Specifies the library where the named modules reside.

- ***module_name***

  Indicates a relocatable object module to be added to the library named in a previous **OPEN** or **CREATE** command. Parentheses (**( )**) are required if individual modules are to be specified.

The **ADDLIB** command adds one or more object modules from one library to the library currently being created or modified.

You can include the entire library by entering the *library_filename* and no module names.

The **OPEN** or **CREATE** command must precede the **ADDLIB** command and name the library to which the object modules will be added.

Related Commands

CREATE, **OPEN**

Example

```
OPEN   LIBRARY1.LIB
ADDLIB MATH.LIB(square.o,sqroot.o)
SAVE
...
```

In this example, **LIBRARY1.LIB** is opened. The **ADDLIB** command adds the modules named **square.o** and **sqroot.o** from the library **MATH.LIB** to **LIBRARY1.LIB**. No changes occur to **MATH.LIB**. If **SAVE** is not entered, no changes will occur to **LIBRARY1.LIB**.

ADDMOD — Adds Object Module(s) to Current Library

Abbreviation

ADDM

Syntax

ADDMOD ***filename***[,***filename***]...

Description

- **filename**

  Specifies the file to be added to the library named in the **OPEN** or **CREATE** command.

The **ADDMOD** command adds a nonlibrary file containing one or more relocatable object modules to the library named in the **OPEN** or **CREATE** command.

The **OPEN** or **CREATE** command must precede the **ADDMOD** command.

Related Commands

CREATE, **OPEN**

Example

```
OPEN   LIBRARY2.LIB
ADDMOD MATH.MBR
SAVE
. . .
```

In this example, the **ADDMOD** command adds the file **MATH.MBR** to the library named in the **OPEN** command, **LIBRARY2.LIB**. The filename, with the path removed, is used as the module name.

CLEAR — Clears Library Session Since Last SAVE

Abbreviation

CL

Syntax

CLEAR

Description

The **CLEAR** command clears all commands that have been entered in the current library session since the last **SAVE** command.

Related Commands

SAVE

Example

```
OPEN   LIBRARY2.LIB
ADDMOD MATH.o
SAVE
CLEAR
OPEN   WRONG_LIB.LIB
ADDMOD FONT.o
CLEAR
OPEN   LIBRARY3.LIB
...
```

In this example, **CLEAR** must be executed before opening and processing **WRONG_LIB.LIB** and **LIBRARY3.LIB**. Note that **WRONG_LIB.LIB** was not modified since no **SAVE** command was executed before the following **CLEAR** command.

CREATE — Defines New Library

Abbreviation

CR

Syntax

CREATE ***library_name***

Description

- **library_name**

  Specifies the file name of the library file being created.

The **CREATE** command defines a new library. Naming conventions should follow those of your host computer and operating system. You can create only one library at a time. A newly created library must be saved before a second one is created.

It is an error to add, replace, delete, or extract modules when no library is opened for modification or creation. It is also an error to use the name of an existing library with the **CREATE** command. If *library_name* exists in the current or indicated directory, a warning message will be issued in interactive mode; in batch mode, an error will be issued, and the new library will not be saved.

The library format that a created library is saved with is set when the library is created, but it can be modified later. The default library format type is to use the Microtec IEEE-695 library format. However, the **FORMAT** command may be used after the library is created to change the library format to SYS V PORTAR. The library format setting only needs to be decided before the library is saved.

Related Commands

OPEN

Example

```
CREATE TEMPOR.LIB
. . .
```

In this example, the command **CREATE TEMPOR.LIB** creates a library file called **TEMPOR.LIB**.

If **TEMPOR.LIB** already exists, a warning is displayed in interactive mode. When you use the librarian in batch or command line mode and you name an existing library with the **CREATE** command, the librarian issues an error message. No library is created.

DELETE — Deletes Module(s) From Current Library

Abbreviation

DE

Syntax

DELETE ***module_name***[,*module_name*]...

Description

- **module_name**

  Specifies the name of the module to be removed from the library named in a previous **OPEN** or **CREATE** command.

The **DELETE** command removes one or more relocatable object modules from the library named in the **OPEN** or **CREATE** command. Object module names are case-sensitive.

An **OPEN** or **CREATE** command must precede **DELETE**.

Related Commands

CREATE, **OPEN**

Example

```
OPEN   LIBRARY3.LIB
DELETE ARCTAN.o,SQUARE.o,RAD.o
SAVE
...
```

In this example, the **DELETE** command deletes relocatable object modules named **ARCTAN.o**, **SQUARE.o**, and **RAD.o** from the library named **LIBRARY3.LIB**.

DIRECTORY — Lists Library Modules

Abbreviation

DI

Syntax

DIRECTORY ***library_name***[(*module_name*[,*module_name*]...)] [*list_filename*]

Description

- **library_name**

  Specifies the name of the library whose module names and sizes are to be listed.

  If you enter only the *library_name*, all modules are listed.

- ***module_name***

  Specifies the name of the module whose size is to be listed. Parentheses are required.

  When you enter specific module names, directory information is displayed for the named modules only.

- ***list_filename***

  Writes the directory information to the named file. If not specified, the output defaults to the standard list device, usually the terminal.

The **DIRECTORY** command lists the format type of the library, module names, and sizes of the modules in the specified library. The sizes listed are the number of bytes required to store the modules on the host computer system.

Object module names are case-sensitive.

Related Commands

FULLDIR

Example

```
DIRECTORY sieve.lib
```

In this example, all modules in **sieve.lib** and their sizes are listed:

```
IEEE-695 Library sieve.lib
Module                Size
SIEVE.o .....      386
MODULE1.o ...      397
MODULE.o ....      289

Module Count = 3
```

END/QUIT — Terminates Librarian Execution

Abbreviation

EN

Q

Syntax

END

QUIT

Description

The **END** or **QUIT** command terminates librarian command processing without building a new library.

If you do not issue a **SAVE** command, the librarian will not build a new library.

Related Commands

SAVE

Example

```
DIR  NEW.LIB
END
```

In this example, the librarian lists the contents of the library **NEW.LIB**. The **END** command exits the library. Since there is no **SAVE** command, a new library is not built.

EXTRACT — Copies Library Module to a File

Abbreviation

EXT

Syntax

EXTRACT ***module_name***[,*module_name*]...

Description

- **module_name**

  Specifies the module to be copied from the library named in a previous **OPEN** or **CREATE** command.

The **EXTRACT** command copies a library module to a file outside the library. The file contains the same information as when it was originally generated by the assembler. Consequently, it can be explicitly loaded. The file has the same name as the module within the library. The extracted module is placed in the current directory.

An **OPEN** or **CREATE** command must precede the **EXTRACT** command.

Related Commands

CREATE, **OPEN**

Example

```
OPEN  libasc.lib
EXTRACT moda.o,modb.o,modc.o
...
```

In this example, the modules **moda.o**, **modb.o**, and **modc.o** are extracted from the current library and written to files with the same names outside the library.

EXTRACTALL — Extracts All Modules From the Current Library

Abbreviation

EXTRACTA

Syntax

EXTRACTALL

Description

The **EXTRACTALL** command copies each library module in the current library to its own object file in the current directory. The file is given the same name as the module name within the library. This command is equivalent to specifying all modules in the current library with the **EXTRACT** command. A warning is emitted if the current library contains no modules.

An **OPEN** or **CREATE** command must precede the **EXTRACTALL** command.

Related Commands

CREATE, **EXTRACT**, **OPEN**

Example

```
OPEN  libasc.lib
EXTRACTALL
```

In this example, all modules within **libasc.lib** will be extracted to files with the same names in the current directory.

FORMAT — Sets Output Library Type

Abbreviation

FO

Syntax

FORMAT { IEEE | PORTAR }

Description

The **FORMAT** command indicates what format the librarian should save the present library in. The two possibilities are Microtec IEEE-695 and SYS V PORTAR library formats. If a library is being created, the default is to create the library using the Microtec IEEE-695 format. This command can be used to set the format type to SYS V PORTAR. Likewise, if an existing library is being modified, this command can be used to change the format from one type to the other. The format change only occurs if the library is saved.

Besides affecting the format of the current library, the **FORMAT** command also changes the default library output format type until the next **FORMAT** command is executed or until the end of the library session. Note that modifying existing libraries will not affect their output type unless a **FORMAT** command is executed after that library is opened. The **-c** and **-p** command line options are equivalent to executing a **FORMAT IEEE** or **FORMAT PORTAR** after the library on the command line is opened or created. Those options can therefore be used to modify the format type of the indicated library.

The Microtec librarian cannot be used to create a stripped SYS V PORTAR library. Any library created or modified by the librarian will be written with a symbol table. The librarian is able to read SYS V PORTAR libraries that have been stripped.

Related Commands

CREATE, **OPEN**, **SAVE**

Example

The following example creates a new library and saves it in SYS V PORTAR format.

```
create libasc.lib
format portar
addmod cpufmc.o
save
```

FULLDIR — Displays Library or Library Module Contents

Abbreviation

FU

Syntax

FULLDIR ***library_name***[(*module_name*[,*module_name*]...)] [*list_filename*]

Description

- **library_name**

  Specifies the library file whose contents are to be listed.

  If you enter just the *library_name*, the contents of all modules are listed.

- ***module_name***

  Specifies the module whose contents are to be listed. Parentheses are required.

  When you enter specific *module_names*, information is displayed for the named modules only.

- ***list_filename***

  Writes the directory display information to the named file. If not specified, the output defaults to the standard list device, usually the terminal.

The **FULLDIR** command provides a full directory display of a library's contents including library format type, module names, their sizes, all global symbol definitions, and external symbol references. The sizes listed are the number of bytes required to store the modules on the host computer system.

Related Commands

DIRECTORY

Example

```
FULLDIR trig.lib (T1.o, T2.o) trig.lst
...
```

In this example, the librarian displays information about modules **T1.o** and **T2.o**, which are members of the library named **trig.lib**. The output listing is written to the file named **trig.lst** as follows:

```
IEEE-695 Library trig.lib

 Module        Size
T1.o ...       414

       ****** PUBLIC DEFINITIONS ******
vara
varb
```

```
       ****** EXTERNAL REFERENCES ******
var1
var2
var3

Public Count   = 2
External Count = 3

 Module                              Size
T2.o ...      785

       ****** PUBLIC DEFINITIONS ******
pub1

Public Count   = 1
External Count = 0
Module Count   = 2
```

HELP — Displays Context-Sensitive Command Syntax

Abbreviation

H

Syntax

HELP

Description

The **HELP** command lists commands with their correct invocation syntax. **HELP** is context-sensitive; the commands displayed are only those that can be legally entered at the time you type **HELP**.

Related Commands

None

Example

```
libppc> help
   CLEAR
   CREATE library_name
   DIRECTORY library_name[(module_name[,...])] [list_filename]
   END
   FORMAT [IEEE | PORTAR]
   FULLDIR library_name[(module_name[,...])] [list_filename]
   HELP
   LIST library_name[(module_name[,...])] [output_file]
   OPEN library_name
   SAVE

libppc> open rp005.lib

libppc> help
   ADDLIB library_name[(module_name[,...])]
   ADDMOD filename[,...]
   CLEAR
   DELETE module_name[,...]
   DIRECTORY library_name[(module_name[,...])] [list_filename]
   END
   EXTRACT module_name[,...]
   EXTRACTALL
   FORMAT [IEEE | PORTAR]
   FULLDIR library_name[(module_name[,...])] [list_filename]
   HELP
   LIST library_name[(module_name[,...])] [output_file]
   REPLACE filename[,filename]
   SAVE

libppc> end
```

OPEN — Opens an Existing Library

Abbreviation

O

Syntax

OPEN ***library_name***

Description

- **library_name**

  Specifies the name of the library file to be opened.

The **OPEN** command lets an existing library be referenced in conjunction with succeeding commands that add, delete, replace, or extract modules. Only one library can be opened at a time.

If you create a new version of the library by modifying an existing library, the updated library will have the same name as the current library. On Windows, the original version of the file will be renamed to have a **.bak** extension in the same directory as the original file.

If the library cannot be located or opened for input, a warning is reported and a new library is created.

When an opened library is saved, it is normally saved with the same library format that it had originally. This can be changed by using the **FORMAT** command to specify the library format that you would like the library to be saved with.

Related Commands

CREATE

Example

```
OPEN MATH.LIB
...
```

In this example, the library named **MATH.LIB** is opened.

REPLACE — Replaces Library Module

Abbreviation

R

Syntax

REPLACE ***filename***[,*filename*]...

Description

- **filename**

  Specifies a file containing one or more modules that will replace the module of the same name in the library named in the **OPEN** or **CREATE** command.

The **REPLACE** command replaces one or more library modules with one or more nonlibrary object modules with the same name. The replacement object module must have the same name as the library module it replaces.

REPLACE mod1.o is not the same as **DELETE mod1.o** followed by **ADDMOD mod1.o**. **ADDMOD** always puts the new module at the end of the library whereas **REPLACE** retains the original module order. If the module does not already exist in the library, a warning is issued, and the module is appended to the end of the library.

REPLACE must be preceded by an **OPEN** or **CREATE** command.

Related Commands

ADDMOD, **CREATE**, **DELETE**, **OPEN**

Example

```
OPEN    LIBRARY1.LIB
REPLACE sentin.o,modu1.o
SAVE
...
```

In this example, the **OPEN** command opens the library **LIBRARY1.LIB**. The library modules **sentin.o** and **modu1.o** are replaced by non-library modules of the same name.

SAVE — Saves Contents of Current Library

Abbreviation

S

Syntax

SAVE

Description

The **SAVE** command saves the contents of the library being created or modified. During this time, the **ADDMOD**, **ADDLIB**, **DELETE**, and **REPLACE** commands are processed. Although these commands were checked for correct syntax and module existence at the time they were entered, the specified modules are not added, deleted, or replaced until a **SAVE** command is issued.

If you are saving to an existing library file in Windows, the original version of the library is saved with a **.bak** extension in the same directory as the original library.

By default, newly-created libraries will be saved in Microtec IEEE-695 format while existing libraries are saved in the same format that they originally existed in. The format type can be selected with the **FORMAT** command.

Related Commands

END

Example

```
CREATE NEW.LIB
ADDMOD REL1.o, REL2.o
ADDMOD FORTUN.o
SAVE
. . .
```

In this example above, **NEW.LIB** is a newly created library comprising of the relocatable object modules named **REL1.o**, **REL2.o**, and **FORTUN.o**.

Chapter 15
Sample Librarian Session

The sample librarian test programs and output listings in this chapter show examples of the input command file and the output listing format.

During interactive program execution, the information is displayed on the terminal. When the librarian is executed in batch mode, the information is displayed in an output stream formatted in the same way as the output listings shown.

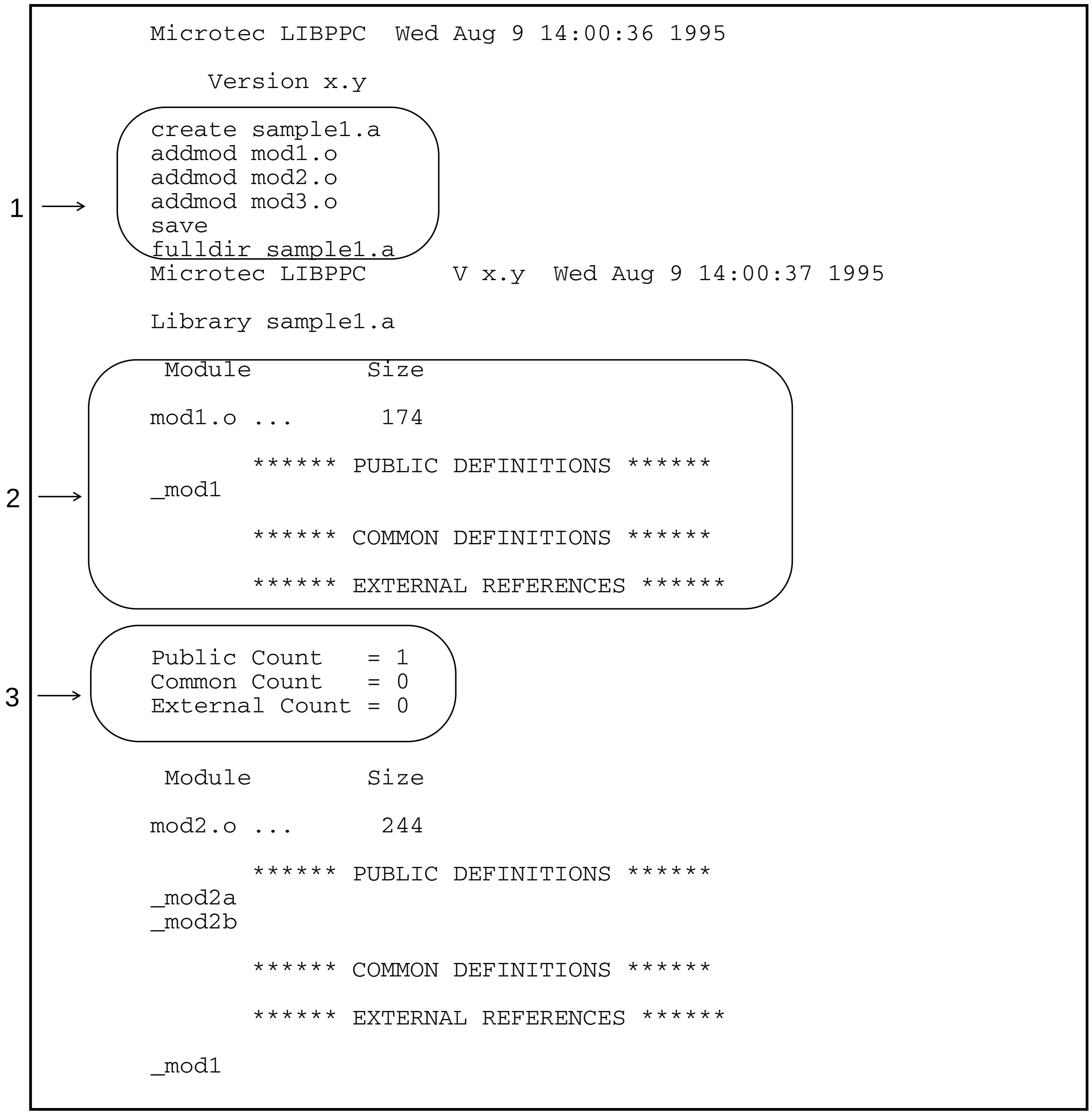
Librarian Sample Program 1

Sample Program 1 is shown in Figure 15-1. Refer to the following points in the text and the sample listing:

1. A new library, **sample1.a**, is created. The command file is shown in the listing. Note that the **END** command is shown at the bottom of the listing. Three modules (**mod1.o**, **mod2.o**, and **mod3.o**) are added to the library. The contents of the library are then listed with the **fulldir** command.
2. The output listing shows each module name, the module’s public definitions, its common definitions, and its external references.
3. The total public symbol count, total common symbol count, and total external symbol count are listed for each module.
4. The total module count is displayed at the end of the listing and the number of warnings or errors encountered.

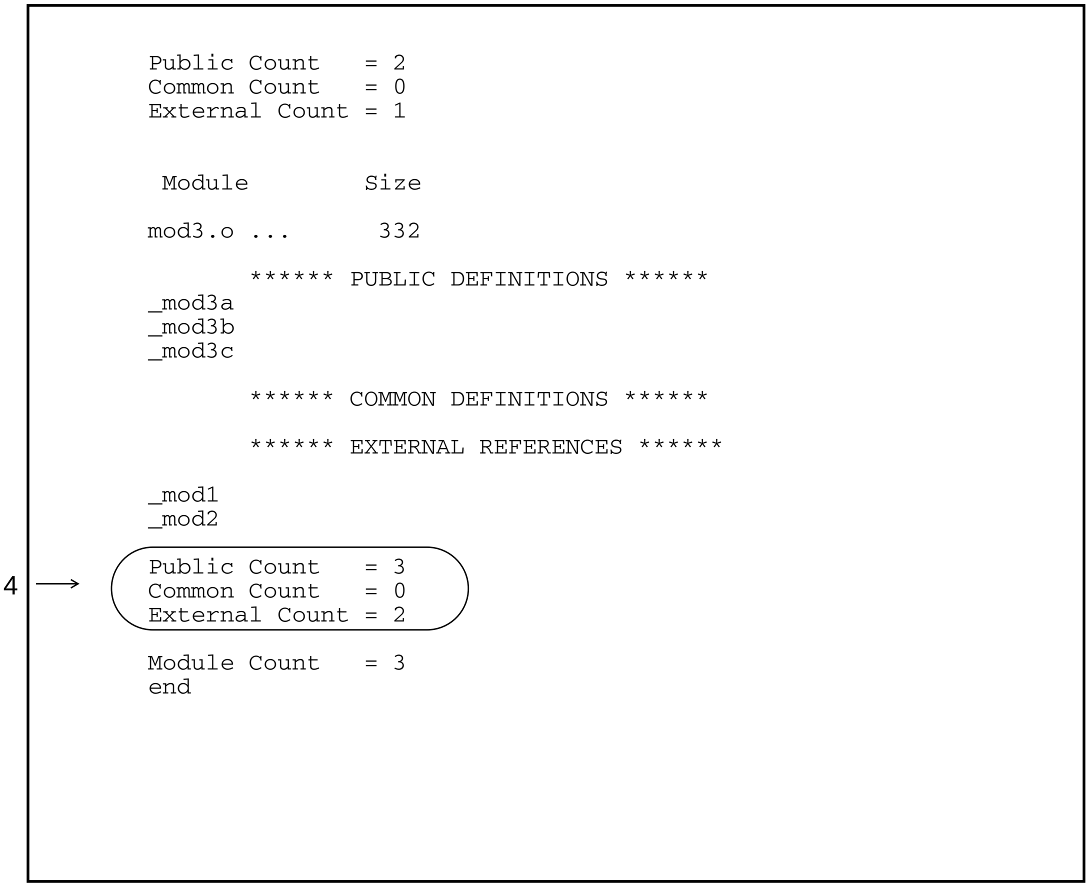
Sample Program 1 Output Listing

Figure 15-1. Librarian Sample Program 1 Output Listing



```
Microtec LIBPPC  Wed Aug 9 14:00:36 1995

    Version x.y

create sample1.a
addmod mod1.o
addmod mod2.o
addmod mod3.o
save
fulldir sample1.a
Microtec LIBPPC      V x.y  Wed Aug 9 14:00:37 1995

Library sample1.a

 Module         Size

mod1.o ...       174

       ****** PUBLIC DEFINITIONS ******
_mod1

       ****** COMMON DEFINITIONS ******

       ****** EXTERNAL REFERENCES ******

Public Count   = 1
Common Count   = 0
External Count = 0

 Module         Size

mod2.o ...       244

       ****** PUBLIC DEFINITIONS ******
_mod2a
_mod2b

       ****** COMMON DEFINITIONS ******

       ****** EXTERNAL REFERENCES ******

_mod1
```

Figure 15-2. Librarian Sample Program 1 Output Listing (cont.)



```
Public Count   = 2
Common Count   = 0
External Count = 1


 Module        Size

mod3.o ...      332

       ****** PUBLIC DEFINITIONS ******
_mod3a
_mod3b
_mod3c

       ****** COMMON DEFINITIONS ******

       ****** EXTERNAL REFERENCES ******

_mod1
_mod2

Public Count   = 3
Common Count   = 0
External Count = 2

Module Count   = 3
end
```

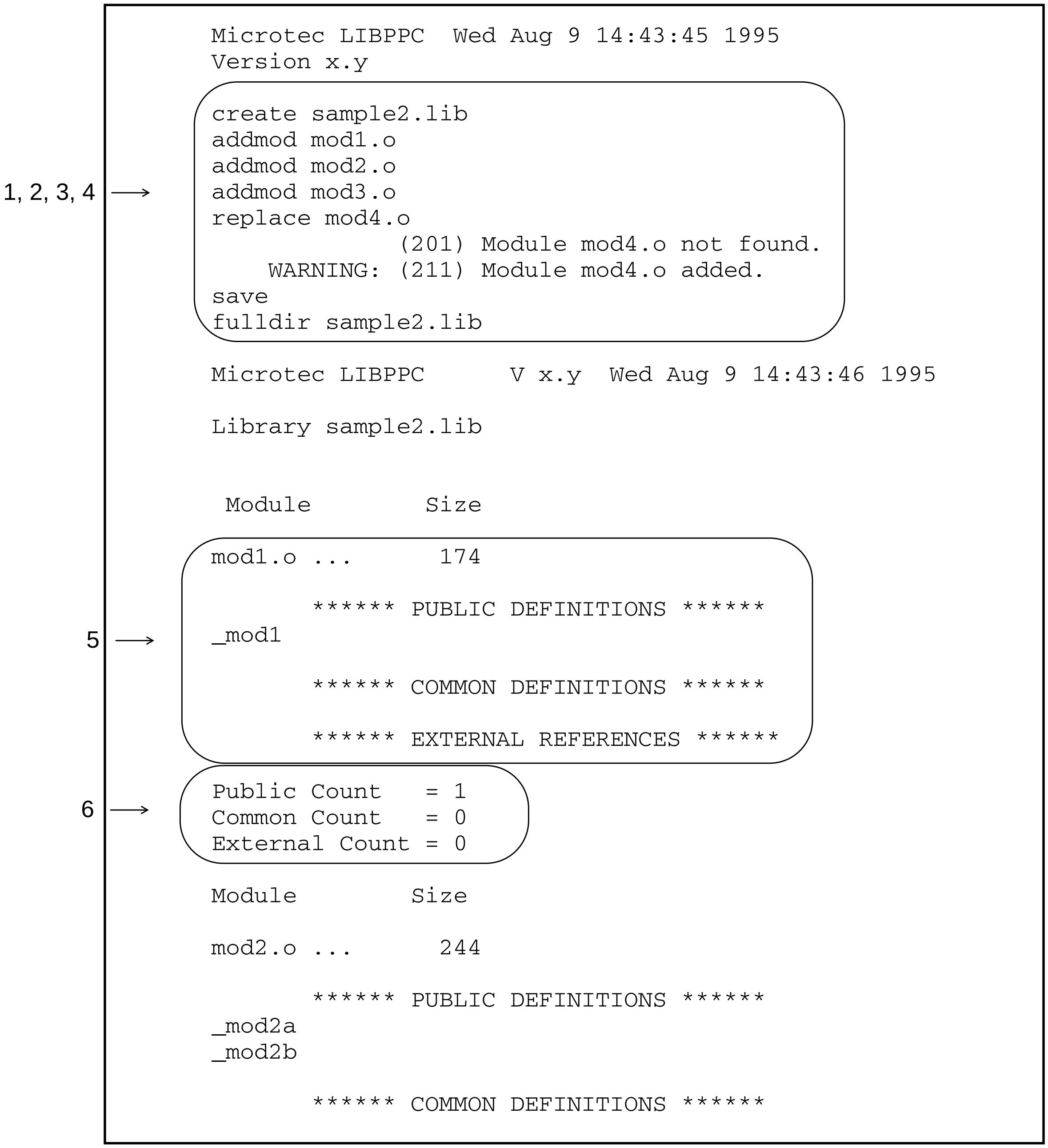
Librarian Sample Program 2

Sample Program 2 is shown in Figure 15-3. Refer to the following points in the text and the sample listing:

1. A new library, **sample2.lib**, is created. The command file is shown in the listing.
2. The **addmod** command is used to add three modules: **mod1.o**, **mod2.o**, and **mod3.o**.
3. The **replace** command attempts to replace module **mod4** with the module **mod4.o**. However, because the module **mod4** does not exist, a warning is issued and **mod4** is added.
4. The contents of the library are then listed with the **fulldir** command.
5. The listing shows each module name, its public definitions, its common definitions, and its external references.
6. The total public symbol count, total common symbol count, and total external symbol count are listed for each module.
7. The total module count as well as total warnings and errors are displayed at the end of the listing.

Sample Program 2 Output Listing

Figure 15-3. Librarian Sample Program 2 Output Listing



```
Microtec LIBPPC  Wed Aug 9 14:43:45 1995
Version x.y

create sample2.lib
addmod mod1.o
addmod mod2.o
addmod mod3.o
replace mod4.o
             (201) Module mod4.o not found.
    WARNING: (211) Module mod4.o added.
save
fulldir sample2.lib

Microtec LIBPPC      V x.y  Wed Aug 9 14:43:46 1995

Library sample2.lib


 Module        Size

mod1.o ...      174

       ****** PUBLIC DEFINITIONS ******
_mod1

       ****** COMMON DEFINITIONS ******

       ****** EXTERNAL REFERENCES ******

Public Count   = 1
Common Count   = 0
External Count = 0

Module        Size

mod2.o ...      244

       ****** PUBLIC DEFINITIONS ******
_mod2a
_mod2b

       ****** COMMON DEFINITIONS ******
```

Figure 15-4. Librarian Sample Program 2 Output Listing (cont.)

```
       ****** EXTERNAL REFERENCES ******

_mod1

Public Count   = 2
Common Count   = 0
External Count = 1


 Module         Size

mod3.o ...       332

       ****** PUBLIC DEFINITIONS ******
_mod3a
_mod3b
_mod3c

       ****** COMMON DEFINITIONS ******

       ****** EXTERNAL REFERENCES ******

_mod1
_mod2

Public Count   = 3
Common Count   = 0
External Count = 2


 Module         Size

mod4.o ...       340

       ****** PUBLIC DEFINITIONS ******

_mod4a

       ****** COMMON DEFINITIONS ******

       ****** EXTERNAL REFERENCES ******


Public Count   = 1
Common Count   = 0
External Count = 0
```

Figure 15-5. Librarian Sample Program 2 Output Listing (cont.)

```
        Module Count   = 4
        end
7 →
        Warnings = 1
        Errors   = 0
```

Appendix A
ASCII Character Set

Table A-1. ASCII Character Set

	Numeric Value			
Name	**Octal**	**Decimal**	**Hex**	**Description**
NUL	00	0	0	NULL
SOH	01	1	1	Start of heading
STX	02	2	2	Start of text
ETX	03	3	3	End of text
EOT	04	4	4	End of transmission
ENQ	05	5	5	Inquiry
ACK	06	6	6	Acknowledge
BEL	07	7	7	Bell
BS	010	8	8	Backspace
HT	011	9	9	Horizontal tab
LF	012	10	A	Line feed
VT	013	11	B	Vertical tab
FF	014	12	C	Form feed
CR	015	13	D	Carriage return
SO	016	14	E	Shift out
SI	017	15	F	Shift in
DLE	020	16	10	Data link escape
DC1	021	17	11	Device control 1
DC2	022	18	12	Device control 2
DC3	023	19	13	Device control 3
DC4	024	20	14	Device control 4
NAK	025	21	15	Negative acknowledge
SYN	026	22	16	Synchronous idle

Table A-1. ASCII Character Set (cont.)

	Numeric Value			
Name	**Octal**	**Decimal**	**Hex**	**Description**
ETB	027	23	17	End of block
CAN	030	24	18	Cancel
EM	031	25	19	End of medium
SUB	032	26	1A	Substitute
ESC	033	27	1B	Escape
FS	034	28	1C	File separator
GS	035	29	1D	Group separator
RS	036	30	1E	Record separator
US	037	31	1F	Unit separator
SP	040	32	20	Space
!	041	33	21	Exclamation mark
"	042	34	22	Double quote
#	043	35	23	Pound sign
$	044	36	24	Dollar sign
%	045	37	25	Percent sign
&	046	38	26	Ampersand
’	047	39	27	Apostrophe / Single quote
(	050	40	28	Left parentheses
)	051	41	29	Right parentheses
*	052	42	2A	Asterisk
+	053	43	2B	Plus
,	054	44	2C	Comma
-	055	45	2D	Hyphen/Minus
.	056	46	2E	Period / Decimal point / Dot
/	057	47	2F	Slash
0	060	48	30	Zero
1	061	49	31	One
2	062	50	32	Two
3	063	51	33	Three

Table A-1. ASCII Character Set (cont.)

	Numeric Value			
Name	Octal	Decimal	Hex	Description
4	064	52	34	Four
5	065	53	35	Five
6	066	54	36	Six
7	067	55	37	Seven
8	070	56	38	Eight
9	071	57	39	Nine
:	072	58	3A	Colon
;	073	59	3B	Semicolon
<	074	60	3C	Less than / Left angle bracket
=	075	61	3D	Equals sign
>	076	62	3E	Greater than / Right angle bracket
?	077	63	3F	Question mark
@	0100	64	40	“At” sign
A	0101	65	41	Upper-case A
B	0102	66	42	Upper-case B
C	0103	67	43	Upper-case C
D	0104	68	44	Upper-case D
E	0105	69	45	Upper-case E
F	0106	70	46	Upper-case F
G	0107	71	47	Upper-case G
H	0110	72	48	Upper-case H
I	0111	73	49	Upper-case I
J	0112	74	4A	Upper-case J
K	0113	75	4B	Upper-case K
L	0114	76	4C	Upper-case L
M	0115	77	4D	Upper-case M
N	0116	78	4E	Upper-case N
O	0117	79	4F	Upper-case O

Table A-1. ASCII Character Set (cont.)

	Numeric Value			
Name	Octal	Decimal	Hex	Description
P	0120	80	50	Upper-case P
Q	0121	81	51	Upper-case Q
R	0122	82	52	Upper-case R
S	0123	83	53	Upper-case S
T	0124	84	54	Upper-case T
U	0125	85	55	Upper-case U
V	0126	86	56	Upper-case V
W	0127	87	57	Upper-case W
X	0130	88	58	Upper-case X
Y	0131	89	59	Upper-case Y
Z	0132	90	5A	Upper-case Z
[	0133	91	5B	Left square bracket
\	0134	92	5C	Backslash
]	0135	93	5D	Right square bracket
^	0136	94	5E	Caret
_	0137	95	5F	Underscore
`	0140	96	60	Backquote
a	0141	97	61	Lower-case a
b	0142	98	62	Lower-case b
c	0143	99	63	Lower-case c
d	0144	100	64	Lower-case d
e	0145	101	65	Lower-case e
f	0146	102	66	Lower-case f
g	0147	103	67	Lower-case g
h	0150	104	68	Lower-case h
i	0151	105	69	Lower-case i
j	0152	106	6A	Lower-case j
k	0153	107	6B	Lower-case k
l	0154	108	6C	Lower-case l

Table A-1. ASCII Character Set (cont.)

	Numeric Value			
Name	Octal	Decimal	Hex	Description
m	0155	109	6D	Lower-case m
n	0156	110	6E	Lower-case n
o	0157	111	6F	Lower-case o
p	0160	112	70	Lower-case p
q	0161	113	71	Lower-case q
r	0162	114	72	Lower-case r
s	0163	115	73	Lower-case s
t	0164	116	74	Lower-case t
u	0165	117	75	Lower-case u
v	0166	118	76	Lower-case v
w	0167	119	77	Lower-case w
x	0170	120	78	Lower-case x
y	0171	121	79	Lower-case y
z	0172	122	7A	Lower-case z
{	0173	123	7B	Left curly brace
|	0174	124	7C	Vertical bar
}	0175	125	7D	Right curly brace
~	0176	126	7E	Tilde
DEL	0177	127	7F	Delete

Appendix B
Assembler Error Messages

This appendix describes the error messages and warnings that appear if errors in the source program are detected during the assembly process. The error message is printed on the listing immediately following the statement in error.

In the event of an error while processing an instruction, an effective NOP (that is, a branch to the next instruction) is generated and processing continues on the next line. If the assembler detects an error while processing a directive, no code is generated and processing continues on the next line. The error message is indicated on the line below the error.

The next section lists assembler messages with a description. Most error messages are self-explanatory. In all cases, unless otherwise noted, they represent error conditions that should be fixed before proceeding to the linker. Warning messages should be checked to verify that the assembler made the right assumptions prior to linking.

Note

Messages that are outside the scope of this product (for example, operating system error messages) are not documented in this appendix. Refer to the *Release Notes* for additional information on error messages.

If you encounter an undocumented error, please contact Technical Support with the exact text of the message and the circumstances under which it occurred. The Technical Support representative will assist you in determining the cause of the problem.

Assembler Messages and Errors

500 **No error**

501 **Missing argument**

The assembler expected an argument to be provided for the statement. Check the syntax of the statement.

502 **Operator expected**

The assembler expected an operator but none was provided. Check the syntax of the statement.

503 **A symbol was found which is invalid in this context**

A register name or a symbol was found when it was not expected. You may have specified the wrong keyword for a directive. Check this manual for the correct symbol or character.

504 Right parentheses not valid in this context

A right parentheses was included in a statement but not expected. Check the syntax of the statement. See Chapter 4, “Assembly Instructions”, for examples of correct use of parentheses.

505 Operator not valid in this context

An operator was used in a statement where it is not valid. Check the syntax of the statement. See Chapter 3, “Assembly Language”, for more information on operators.

507 Operand value must be 0.

This instruction has special operand requirements that the operand may only have a value of 0. Check the hardware manual for a description of this instruction.

508 Unbalanced parentheses

The number of left-parentheses (**(**) in an expression does not match the number of right-parentheses (**)**). The assembler ignores the line containing the expression and prints out an error message.

509 Relocatable expression too complex

This expression is too complex. Check the legality of the expression for the directive. The expression probably contains multiple relocatable symbols or an illegal operation on a relocatable symbol.

510 Stack underflow (internal error)

The expression is probably too complicated. Make the expression shorter by substituting a series of simpler expressions for the longer, more complex one.

511 Invalid operands for !! operator

The operands must be absolute. Specifically, the operand must be either a constant or an expression that is evaluated to an absolute value by the assembler.

512 Invalid operands for && operator

The operands must be absolute. Specifically, the operand must be either a constant or an expression that is evaluated to an absolute value by the assembler.

513 Invalid operands for || operator

The operands must be absolute. Specifically, the operand must be either a constant or an expression that is evaluated to an absolute value by the assembler.

514 Invalid operands for ^ operator

The operands must be absolute. Specifically, the operand must be either a constant or an expression which is evaluated to an absolute value by the assembler.

515 Invalid operands for == operator

The operands must be absolute. Specifically, the operand must be either a constant or an expression that is evaluated to an absolute value by the assembler.

516 Invalid operands for != operator

The operands must be absolute. Specifically, the operand must be either a constant or an expression that is evaluated to an absolute value by the assembler.

517 Invalid operands for >= operator

The operands must be absolute. Specifically, the operand must be either a constant or an expression that is evaluated to an absolute value by the assembler.

518 Invalid operands for > operator

The operands must be absolute. Specifically, the operand must be either a constant or an expression that is evaluated to an absolute value by the assembler.

519 Invalid operands for < operator

The operands must be absolute. Specifically, the operand must be either a constant or an expression that is evaluated to an absolute value by the assembler.

520 Invalid operands for <= operator

The operands must be absolute. Specifically, the operand must be either a constant or an expression that is evaluated to an absolute value by the assembler.

521 Invalid operands for >> operator

The operands must be absolute. Specifically, the operand must be either a constant or an expression that is evaluated to an absolute value by the assembler.

522 Invalid operands for << operator

The operands must be absolute. Specifically, the operand must be either a constant or an expression that is evaluated to an absolute value by the assembler.

523 Invalid operands for * operator

The operands must be absolute. Specifically, the operand must be either a constant or an expression that is evaluated to an absolute value by the assembler.

524 Invalid operands for / operator

The operands must be absolute. Specifically, the operand must be either a constant or an expression that is evaluated to an absolute value by the assembler.

525 Invalid character (c)

An unexpected character or a character that is not in the set supported by the assembler was found. Check this manual to determine if the character is supported.

526 Closing string delimiter missing

A string must be enclosed in double quotation marks (" "). An opening quotation mark was entered without a closing quotation mark.

527 String invalid in this context

A double quote character (") appeared where it is not valid. Remove this character to fix the error.

528 Invalid opcode

An opcode that the assembler does not recognize was entered. Check the opcode against the list provided in Chapter 4, “Assembly Instructions”.

529 Invalid operands for ~ operator

The operands must be absolute. Specifically, the operand must be either a constant or an expression that is evaluated to an absolute value by the assembler.

530 Undefined symbol: symbol

A symbolic name in the operand field that was found was never defined. The symbol should have been previously defined for certain directives but was not, although the symbol might have been defined after the directive. If the symbol has not been defined anywhere in the program, fix the error by defining the symbol.

531 Invalid nesting of .IF...ENDIF

An **.endif** was used without the preceding **.if** directive. Check that every **.endif** conditional statement terminator has a matching **.if** directive.

532 Invalid nesting of .IF...ELSE...ENDIF

An **.else** and **.endif** was used without the preceding **.if** directive. Check that every **.else** and **.endif** directive has a matching **.if**.

533 Missing .ENDIF

Self-explanatory.

534 .IF stack overflow; limit is 15 nesting levels

The assembler allows a maximum of 15 nesting levels for the **.if** conditional directive. Fix the error by reducing the number of nested **.if**s.

535 Invalid operands for & operator

The operands must be absolute. Specifically, the operand must be either a constant or an expression that is evaluated to an absolute value by the assembler.

536 Invalid operands for | operator

The operands must be absolute. Specifically, the operand must be either a constant or an expression that is evaluated to an absolute value by the assembler.

537 Integer value is outside of its legal range

An integer value accompanying an instruction operand or directive operand was found that was outside the range for that particular instruction or directive. Check this manual or the manufacturer's manual to determine the legal range.

538 Name required as first argument on this directive

A directive was found that required a name following the directive. Check Chapter 6, "Assembler Directives", to determine what constitutes a legal name for this particular directive.

539 .FILE source name truncated

A **.file** string was specified that was longer than its maximum limit. Check Chapter 6, "Assembler Directives", to determine the maximum file name size. The name has been truncated to that length.

540 Relocatable expression invalid in this context

The instruction contained an operand that violated a rule of relocation. Specifically, an operand that should have been absolute was being used as a relocatable expression. Fix the error by making the operand an absolute expression.

541 Comma expected

The assembler expected a comma in a statement, but none was provided. Check the syntax of the statement that caused the error.

542 Invalid section name

Either a section name did not follow the rules for an assembler label, was a previously defined symbol, or a **sizeof()**, **sizeha()**, **sizehi()**, **sizelo()**, **startof()**, **startha()**, **starthi()**, or **startlo()** operator did not directly encounter a section name. Check Chapter 3, "Assembly Language", for the rules governing the naming of a section.

544 Macro definition or repeat block terminated by assembler

An end of file condition was detected while processing a macro definition or repeat block. A **.mend** or **.endrept** directive was automatically added to end the definition. Processing continues.

545 Too many sections

A maximum of 200 sections, including **.dsect** sections, are allowed. The **.sect** or **.dsect** directive is ignored. Fix the error by reducing the number of sections or by splitting up the assembly source file.

546 Invalid local symbol: name

The symbol indicated is neither a valid symbol nor a recognized predefined symbol. Check Chapter 3, “Assembly Language”, for valid forms of these symbol types.

547 This symbol cannot be made global/external: name

The symbol cannot be made globally visible to other modules for some reason, such as the symbol being a section name or not being a valid symbol. Fix the error by not allowing the symbol to appear in a **.extern** or **.globl** directive. The symbol has already been declared as an section.

548 This symbol cannot be made external: symbol

An attempt was made to make a symbol visible to other modules that cannot be represented in ELF object files appropriately. This symbol cannot be made global.

550 Unable to open Include file

The assembler could not locate the file to be included. Make sure you have entered the file name correctly in the **.include** directive or on the command line.

551 Invalid character in formal parameter list: string

The rules surrounding the definition of formal parameters for a macro have been violated. Check Chapter 3, “Assembly Language”, for the correct method of defining a formal parameter name.

553 Duplicate label or name

The label in the statement has previously been defined. This error can also be caused by a symbol being defined in the **.extern** or **.globl** directive and appearing in the label field of some statement. Fix the error by removing the label from the statement’s label field.

554 Incompatible usage: label not permitted on this directive

The assembler found a label that was not permitted for this particular directive. Check Chapter 3, “Assembly Language”, for the correct syntax regarding this directive.

555 Target not aligned on a word boundary

The target of a branch instruction is not at a word boundary. Branch targets should be other instructions and must be aligned on word boundaries. Check to see that the branch target is legitimate.

557 Unknown or missing option flag

You have specified an option for the **.lflags** directive that the assembler cannot recognize. Check Chapter 6, “Assembler Directives”, for a list of options that the directive allows.

558 Error while writing cache file.

An error occurred while the assembler was trying to create a cache file for the current processor variant. This file is normally created in the dvs/cache directory, in the installation directory. There may a file protection problem with respect to creating either the 'cache' directory or creating files within it.

566 Invalid register expression

A general purpose register was encountered as part of a complex expression. Register names can only be used by themselves.

571 Colon expected

A label without a colon (**:**) was found. This message is a warning. Terminate all label fields with a colon.

587 This instruction has too many operands

The assembler has found a mnemonic that has too many operands. Check Chapter 3, “Assembly Language”, to determine the number of operands allowed for the particular instruction.

591 .ERROR directive assembled

This usually indicates an error condition that you have defined. Normally this directive is used to indicate an invalid macro call. Determine why the assembler assembled this source line.

593 This directive invalid outside a macro

A **.MEXIT** or **.MEND** directive was encountered outside of a macro or a **.ENDREPT** directive was encountered outside of a repeat block. As these directives have no other meaning, you should determine why these statements were assembled. There might be a missing **.MACRO** or **.REPT** directive.

594 This character is used invalidly within real constant: string

An invalid character was found while the assembler was reading a floating-point constant. Check Chapter 3, “Assembly Language”, for the definition of a floating-point constant.

595 A real constant was expected here

Floating-point directives require floating-point or integer constants as operands or parameters.

596 Real numbers invalid in this context

Floating-point constants can only be used as operands for floating-point directives. They cannot be used within arithmetic expressions.

597 This real number is too small to represent. Zero substituted

Self-explanatory. This is only a warning.

598 This real number is too large to represent. Infinity substituted

Self-explanatory. This is only a warning.

599 Macros/include files nested too deeply

Too many levels of macro calls or include files were found. This normally indicates an endless recursive loop. Verify that recursive macro calls are capable of terminating and that the include files do not include each other, or increase the maximum macro recursion depth with the **.mdepth** directive.

600 Invalid use of real constant

A real constant was used in a calculation where real constants cannot be used. Real constants cannot be part of any expression at this time.

601 Value was truncated to fit in its field

An out of range value was encountered. This is a warning only. Specific situations that can cause this error include operand values greater than 255 in a **.byte** directive.

602 Bad displacement value (expecting constant or relocatable)

A displacement expression or value was expected as an expression operand, but a register was found.

605 Cannot obtain dynamic memory

The assembler was unable to allocate memory for storage purposes. More dynamic memory must be made available in order for the assembler to run to completion. Other possible actions might be to not produce a listing or to break the assembly file into smaller pieces.

607 End of File inside a macro or repeat definition

An unterminated macro definition or repeat block was encountered. This is a warning only, as the **end-of-file** marker implicitly terminates the macro definition or repeat block. However, it is likely that a missing **.mend** or **.endrept** directive is causing the undesired results.

608 Invalid operands for % operator

The operands must be absolute. Specifically, the operand must be either a constant or an expression that is evaluated to an absolute value by the assembler.

609 Number out of valid range

An invalid number was encountered. A negative repeat count for an operand in a data directive (for example, **.byte [-1]0**) is the probable cause for this error.

610 Character string (double quote) expected

A character string, delimited by double quotes (`" "`), was expected.

611 Identifier expected

An identifier symbol is missing.

613 Code generation in .DSECT, .BSS, or noload section is illegal

An instruction was encountered in a dummy section (**.dsect**) or a section of type **BSS**.

615 Invalid alignment value

An illegal alignment value was encountered in a **.align** directive.

616 Symbol must be a macro name: name

The identifier list of a **.mundef** directive must contain only previously defined macros. Check the spelling and case of identifiers in the identifier list.

620 Macro expansion overflowed buffer. Truncated

The maximum size of a line in a macro after a macro expansion has occurred is 32768 characters. Excess characters are ignored. This is only a warning. However, it is likely to cause other error conditions.

621 String is too long

The maximum string size of 8192 bytes was exceeded. Verify that the closing string delimiter exists or that the string does not exceed the documented limits.

622 Integer number expected

Self-explanatory.

624 Not aligned on word boundary

An instruction or data directive was assembled on a nonword boundary. Precede this directive with a **.align** directive. This error is a warning. If the error is on an instruction, it will not execute correctly on a PowerPC processor. If it is data, it may not be accessed efficiently on a PowerPC processor.

625 Not aligned on halfword boundary

A data directive was assembled on a non-half-word boundary. Precede this directive with a **.align** directive. This is a warning, but the resulting data may not be accessed efficiently on a PowerPC processor.

627 Text following .END directive ignored

Nonwhite space text follows the **.end** directive. Ensure that the **.end** directive was not encountered prematurely and remove the excess text. This is a warning.

628 Missing right square bracket

The right square bracket used to terminate a repeat count is either missing or improperly positioned.

630 Redefining register symbol as an actual symbol; check uses of this symbol

A symbol was defined that has the same name as a predefined register symbol. Any use of that symbol before this definition will be considered to be a register while any use after this definition will be considered the type of symbol that was just defined. Care must be taken to guarantee that no unexpected behavior occurs. If necessary, the **.lflags nr** directive may be used to turn off the recognition of all predefined register symbols.

634 Unexpected text following end of legal statement

A statement should be terminated with a new-line character or a comment field. Something other than the new-line character or comment field was encountered. This is a warning.

635 Identifier is too long. Truncated to 8192 characters

Self-explanatory. This is a warning.

640 Expression is too large

The expression is too long. It should be made shorter by substituting a series of shorter expressions and by using **.set** directives.

643 Cannot output to list file

Errors were encountered while attempting to output to the listing file. The most likely cause is a full disk. Either reassemble without the list option, or ensure that the device receiving the listing has sufficient space. This is a fatal error.

646 Out of memory

Intermediate forms of the listing and object files are kept in virtual memory. If these files are excessively large, a fatal error occurs. Remove the list option, negate some list flags, or reduce the program size to reduce virtual memory requirements. Check all **.rept** directives with large repeat counts and **.space** directives in non-**bss** sections with large byte counts to ensure that they are not excessively large. This is a fatal error.

648 Division by zero

The assembler encountered an absolute expression containing a divide or modulus by zero. Fix the error by replacing the current expression with a valid one.

649 Recursive expression evaluation

An expression was defined that made a direct or indirect reference to itself. This expression cannot be evaluated and must be rewritten to evaluate to a constant value or a simple relocatable expression.

657 Invalid processor type or processor type already set

An invalid processor was indicated. Check Chapter 6, “Assembler Directives”, for the valid types in the **.cputype** directive.

668 Invalid register symbol or number

An inappropriate register was used, either symbolically or numerically. Check the usage of the indicated directive.

673 Right bracket missing

Self-explanatory.

675 Maximum macro recursion depth cannot change during macro call

An attempt was made to change the maximum number of nested macro calls while within a macro call.

676 PC-relative instructions must have relocatable expressions

A PC-relative instruction has either absolute symbols or absolute values. Check Chapter 6, “Assembler Directives”, and Chapter 7, “Section Directives”, for a description of the allowable expressions for assembler directives.

677 Parenthesized expression not valid in this context

An expression can only be delimited with parentheses (**( )**) rather than commas (**,**) if the expression follows the second operand. Use of parentheses as delimiters in any other context is invalid. This error may also indicate a missing operator.

678 A parenthesized expression is required for the second operand

This instruction requires its third argument to be delimited by parentheses instead of commas. Check Chapter 3, “Assembly Language”, for the correct usage of this instruction.

679 Indicated segment alignment less restrictive than previous value

The indicated alignment is less than the previous alignment and will be ignored. This is a warning only.

680 Segment alignment changed to be more restrictive

The specified alignment is larger than the previous alignment and this new value will take effect. This is a warning only.

681 Branch target expression is larger than 16 bits

The value of the branch target expression cannot fit in 16 bits. Check the displacement of the label from the current instructions and rearrange the label so that the displacement will fit in 16 bits.

682 Branch target expression is larger than 26 bits

The value of the branch target expression cannot fit in 26 bits. Check the displacement of the label from the current instructions and rearrange the label so that the displacement will fit in 26 bits.

683 This instruction is only valid in supervisor mode

This instruction can only be used in supervisor mode. This warning may indicate a potential problem that could result in a run-time exception. This warning can be avoided by using the **.supervisoron** and **.supervisoroff** directives to surround code that is executed in supervisor mode. Check the hardware manual for more information.

684 Invalid when first operand is same as third operand

This instruction has special operand requirements and the first and third operands cannot be same. Check the hardware manual for a description of this instruction.

685 Invalid when third operand is zero

This instruction has special operand requirements and the third operand cannot be zero. Check the hardware manual for a description of this instruction.

686 Invalid when both first and third operands are 0

This instruction has special operand requirements and both the first and third operands cannot be zero. Check the hardware manual for a description of this instruction.

687 Invalid BO value for branch instruction

The **BO** value for this branch instruction is invalid. Check the hardware manual for a description of this instruction.

693 A parenthesized expression is not allowed for this instruction

Self-explanatory. Check the hardware manual for a correct syntax of this instruction.

694 Only one .file directive allowed per assembly source file

Only a single symbol table entry is emitted for the **.file** directive, so only one definition is appropriate.

702 Org values must be within the current section and not be smaller

The only valid expressions for the **.org** directive is either a constant expression or a relocatable expression that belongs to the current section. Also, any expression must result in a program counter that is the same or larger than the current program counter.

705 Undefined symbol in instruction operand

The instruction requires values or registers as operands. Check the hardware manual for the correct usage of this instruction.

707 Relocatable expression required in this context

A constant expression was used where only relocatable expressions may occur. Check the usage of this directive or instruction.

711 If statement not terminated inside include file

A **.if** statement must be completely contained within a single source file. It may not be contained across multiple source files, such as through the use of the **.include** directive.

712 Unknown predefined assembler symbol

An undefined assembler symbol was indicated. Check the use of this symbol against the allowed list of predefined assembler symbols. Perhaps the used symbol was not completely uppercase.

713 Undefined debug symbol: symbol

A symbol was used in a debug directive that was not previously defined. Since debug directives are generally emitted by compilers, this is probably the result of an invalid directive. Please contact Mentor Graphics Technical Support for more information.

714 Branch Prediction not possible with this instruction

Only conditional branch instructions may have a plus sign (+) or minus sign (-) appended to them to indicate branch prediction. Check the usage of the indicated instruction. Another possible problem is that a space or tab character is missing in the expression.

715 Section attributes cannot be modified as specified

Most predefined sections may not have their section attributes changed. Only the **.sdata2** section may have its attributes modified to be allocated and writable. Any other attributes are invalid.

716 Undefined symbol or forward reference

A symbol was used whose value was not known at this time. Some directives may require that their expressions be completely known in order to take the appropriate action. The definition of this symbol must be moved or defined before this use.

717 Possible incompatibility with 64-bit processors

A PowerPC compare instruction was defined to expect 64-bit operands. This form of instruction will not work on the PowerPC processors supported by this product.

718 A register in range of registers being updated

The indicated base register in this instruction was contained within the range of registers to be acted upon. This operation is invalid.

719 Invalid Relocatable reference to non-loadable section

A relocation entry was found in a normal section that references a section that is not actually loaded into memory. This type of reference is not valid.

721 Invalid predefined section name

An unknown section was indicated that has a period (**.**) as its first character. This is not allowed. Either a different section name should be used or the spelling of the section name should be checked.

722 Use of SDA operator invalid in this context

The use of the SDA operator is restricted to those PowerPC load and store instructions that have an address register and a 16-bit signed displacement field. The SDA operator can also be used as the immediate field in the following instructions: **addi**, **addic**, **addic.**, **subi**, **subic**, **subic.**, **la**, and **li**. The SDA operator cannot be used in any other context.

724 Endian type can only be BIG or LITTLE

The operand specified must exactly match BIG or LITTLE.

725 Mixed endian condition detected

A condition arose in which the assembler detected the specification of both BIG- and LITTLE-endian byte order. See the *User's Guide* portion of this manual or the **.endian** directive for more information.

726 Expression used for this operator must be VAL1

The type of expression found with the operator does not conform to valid expression syntax. Please refer to Chapter 3, "Assembly Language", and Chapter 5, "Relocation", for valid expressions allowed for these operators. The expression should conform to the VAL1 form.

727 Expression used for this operator must be VAL2

The type of expression found with the operator does not conform to valid expression syntax. Please refer to Chapter 3, "Assembly Language", and Chapter 5, "Relocation", for valid expressions allowed for these operators. Only VAL2 forms are allowed in this context.

728 Invalid expression specified for 32-bit field

The type of expression found with the operator does not conform to valid expression syntax. Please refer to Chapter 3, "Assembly Language", and Chapter 5, "Relocation", for valid expressions allowed for these operators. The expressions used may conform only to VAL16, VAL2, or VAL3.

729 Invalid expression specified for 26-bit branch target

The type of expression found with the operator does not conform to valid expression syntax. Please refer to Chapter 3, "Assembly Language", and Chapter 5, "Relocation", for valid expressions allowed for these operators. Only VAL2 forms are allowed in this context.

730 Invalid expression specified for 16-bit branch target

The type of expression found with the operator does not conform to valid expression syntax. Please refer to Chapter 3, "Assembly Language", and Chapter 5, "Relocation", for valid expressions allowed for these operators. Only VAL2 forms are allowed in this context.

731 Invalid expression specified for 16-bit field

The type of expression found with the operator does not conform to valid expression syntax. Please refer to Chapter 3, "Assembly Language", and Chapter 5, "Relocation", for valid expressions allowed for these operators. Only VAL16 and VAL1 expressions allowed.

732 Invalid section attribute combination

Bad section attributes were specified with the **.sect** directive. This may indicate no attributes were specified, that the specified attributes are incomplete, or that a **bss** section was reopened with different attributes.

733 Option already set - cannot override previous setting

An option was used in a **.lflags** directive that was already set on the command line to a different state. Some options can only be set once per assembly file. Check the use of the options with this directive and verify that the expected behavior has occurred.

734 Can't change default alignment for common symbols after use of .comm

The **b** flag in the **.lflags** directive cannot be used after a **.comm** directive has been used. Otherwise, the default value given to a common symbol may be different than what the customer expects. This flag must appear before any **.comm** directives in the source file.

735 Can't change processor type after instructions have been processed

The **.cputype** directive cannot be set after an instruction has been used. Since this directive may only be used once in a source file, the **.cputype** directive should be emitted at or near the start of the assembly source file. This directive has no impact on data directives.

736 Register number not supported on this processor for writing

The special purpose register value used with this command is not valid for the specified processor type for the assembly file. Either the register value must be changed or a different processor type must be specified.

737 Register name not supported on this processor for writing

The special purpose register symbol used with this command is not valid for the specified processor type for the assembly file. Either the register symbol must be changed or a different processor type must be specified.

738 Unable to define ._PPC_EMB_procflags

An internal assembler error has occurred. Please contact Mentor Graphics Technical Support for more information.

740 Multiple macro parameters have the same name

A macro was defined that had multiple parameters with the same name. This is not allowed since the assembler cannot know which parameter is being referred to within the macro body.

741 Attempt to rename symbol more than once ignored

An attempt was made to rename the same symbol twice. Only the first rename is accepted.

742 Renamed symbol collides with defined symbol

An attempt was made to rename a symbol to a pre-existing name. This is not allowed. The rename is ignored.

744 Too many macro parameters

The number of allowable macro parameters was exceeded. Only 35 separate macro parameters or arguments are allowed.

745 Too many macro arguments

The number of allowable macro arguments was exceeded. Only 35 separate macro parameters or arguments are allowed.

746 Mismatched or illegal use of .merge_end directive

The **.merge_end** directive was misused somehow. Either it doesn't have a matching **.merge_start** directive, is being used in a non-relocatable section, is being used in a different section than what the **.merge_start** directive was used in, or it is being misused in some other related way.

747 Symbol for empty mergeable block not emitted

The region surrounded by the **.merge_start** and **.merge_end** directives does not contain any code or data. This region does not represent anything meaningfully. It is ignored.

748 Merge symbol not created (internal error): symbol

The assembler was unable to create a symbol to represent a mergeable region. An internal assembler error has occurred. Please contact Mentor Graphics Technical Support for more information.

749 Merge symbol already exists (internal error): symbol

Multiple mergeable regions were created with the same names in the same module. This is not allowed. Please contact Mentor Graphics Technical Support for more information.

750 Bad .merge_start directive syntax: text

The **.merge_start** directive may only take a symbol as the first operand and single characters for all other operands. An invalid operand was detected.

751 Previous .merge_start not properly terminated

A **.merge_start** directive was used before an earlier mergeable region was terminated with a **.merge_end** directive. It is not valid to nest mergeable region definitions.

752 Associated mergeable region not found: symbol

A mergeable region was defined to be related to a symbol within the module. This symbol was not found. Please contact Mentor Graphics Technical Support for more information.

753 Label for mergeable block doesn't exist or is outside of block

A mergeable region was defined that did not contain its defining symbol. This is not a valid mergeable region.

754 Label within mergeable block is not at start of block

A mergeable region was defined that did not have its defining symbol placed at the beginning of the mergeable region. Such use is not expected.

755 Mergeable region not closed in at least one section

The end of the source file was detected and mergeable regions exist in at least one section that did not have a terminating **.merge_end** directive. Those mergeable regions will be ignored.

756 Illegal section for .merge_start directive

Mergeable regions are only valid in normal relocatable sections. It is not acceptable to define mergeable regions in **BSS** sections or in dummy sections.

757 Mergeable symbol exceeds internal length limits

The symbol used to describe a mergeable region exceeded the maximum length limitations. The mergeable region cannot be defined.

758 Invalid when first operand is same as second operand

This instruction has special operand requirements and the first and second operands cannot be the same. Check the hardware manual for a description of this instruction.

759 Invalid when second operand is 0

This instruction has special operand requirements and the second operand cannot be zero. Check the hardware manual for a description of this instruction.

760 Invalid when both first and second operands are 0

This instruction has special operand requirements and both the first and second operands cannot be zero. Check the hardware manual for a description of this instruction.

761 Output file has same name as input file: filename

The assembler has detected that the designated output file would overwrite one of its input files. This could be either the main source file or one of the include files. The assembler will detect the use of links as well.

762 Reference to non-loadable section not allowed

A non-loadable section was referenced in a way that is not allowed. The reference is ignored.

763 Global symbol in nonloadable section not emitted

It is not allowable to have a global symbol that belongs to a non-loadable section. This symbol is ignored.

764 Invalid when first operand is 0

This instruction has special operand requirements and the first operand cannot be zero. Check the hardware manual for a description of this instruction.

765 Number of arguments not valid for this instruction

The number of operands used with an instruction does not match what is recognized by the assembler. Check the hardware manual for a description of this instruction.

766 Internal data error for instruction definition

The assembler detected an invalid state while processing the current instruction. This is an internal error. Please report this inconsistency to Mentor Graphics Technical Support.

767 Invalid information for processor variant

The assembler detected an invalid state while trying to process information for the selected PowerPC variant. This is an internal error. Please report this inconsistency to Mentor Graphics Technical Support.

771 Invalid register type information found

The assembler detected an invalid state while processing the current instruction. This is an internal error. Please report this inconsistency to Mentor Graphics Technical Support.

772 Invalid or missing information for processor variant

The assembler detected an invalid state while trying to process information for the selected PowerPC variant. This is an internal error. Please report this inconsistency to Mentor Graphics Technical Support.

773 Register number not supported on this processor for reading

The special purpose register value used with this command is not valid for the specified processor type for the assembly file. Either the register value must be changed or a different processor type must be specified.

774 Register name not supported on this processor for reading

The special purpose register symbol used with this command is not valid for the specified processor type for the assembly file. Either the register symbol must be changed or a different processor type must be specified.

775 Timebase Register not allowed with this instruction

This instruction is not able to use the TBL or TBU SPR registers as an operand. The SPR register name or value should be checked against the hardware manual.

776 Timebase Register required with this instruction

This instruction requires the use of the TBL or TBU SPR registers as an operand. No other SPR register is valid.

800 ELF internal error

Invalid ELF information, such as file type, machine type, and so forth, was encountered while writing an ELF object file.

801 ELF write error

File write error encountered while writing an ELF object file.

802 ELF nomem

No memory available to hold intermediate data while writing an ELF object file.

803 Invalid symbol binding

Invalid symbol binding value encountered while writing an ELF object file.

804 Invalid symbol type

Invalid symbol type encountered while writing an ELF object file.

805 Invalid symbol ordering

Some local symbols appear after global symbols in the ELF symbol table while writing an ELF object file.

806 Invalid relocation type

Invalid relocation type encountered while writing an ELF object file.

807 Section does not exist

Destination section does not exist while writing an ELF object file.

808 Section already exists

Duplicate section creation encountered while writing an ELF object file.

809 Name too long

The length of a symbol name longer than 8192 characters encountered while writing an ELF object file.

Appendix C
Linker Error Messages

This appendix describes the error messages and warnings that appear if errors are detected during the link process. The error message is printed on the listing immediately following the statement in error.

Note

Messages that are outside the scope of this product (for example, operating system error messages) are not documented in this appendix.

If you encounter an undocumented error message, please contact Mentor Graphics Technical Support with the exact text of the message and the circumstances under which it occurred. The Technical Support representative will assist you in determining the cause of the problem.

Message Severity Levels

There are two classes of errors that can occur during linker program execution. The first class is nonfatal and includes warnings, messages, and errors. Processing proceeds after a nonfatal error is reported. The second class is fatal, and processing is abandoned.

Status messages can be one of the types listed in Table C-1.

Table C-1. Linker Message Severity Levels

Type	Severity
Warnings	Not fatal. Verify that the assembler made the appropriate assumptions.
Messages	Informative and often appears in conjunction with another error message.
Errors	Usually fatal, but processing continues to facilitate further error checking.
Fatal	Always fatal. Processing aborts.

Linker Messages and Errors

300 **Invalid object record (FATAL)**

An ELF object file contains an incorrect record.

301 Linker-generated symbol symbol defined elsewhere (WARNING)

A symbol that is normally created by the linker has been previously defined. This warning indicates that the linker will not be assigning a value to that symbol.

302 Expecting a string as an argument: text (ERROR)

A quoted string is normally expected as an argument to this command. See Chapter 11, “Linker Commands”, for more information.

303 Differing sizes for common definitions of symbol: SYMBOL (WARNING)

The linker has detected that a common symbol has been defined more than once, but with different size attributes. While allowed, it may indicate a programming problem where a symbol of a given name has been defined in multiple places with different type definitions.

305 Unable to process section: name of type: string in file: filename (WARNING)

The section specified cannot be processed correctly by the linker. If you have specified the section name, section type, and filename, this warning will also give this information. The section reference will be ignored.

306 Out of memory (FATAL)

This message from the symbol table manager informs you that there is not enough memory to read in the symbol table.

307 Symbol: name in file: filename already defined in file: filename (WARNING)

A symbol has been defined more than once. The linker will use the symbol definition for the first symbol with the same name and issue a warning message each time it finds a duplicate name.

308 Cannot specify section after '*' (ERROR)

In the **ORDER** command, if there is a wildcard character (*****), it must be the last operand. See Chapter 11, “Linker Commands”, for more information on this command.

309 Section type of section: name not supported (ERROR)

The linker does not recognize the section type that it encountered. The section with the illegal type will be ignored. Specifically, it will not be allocated, relocated, or linked.

312 Invalid syntax at: string (ERROR)

A syntax error has been found in the specified command. The operand has an invalid syntax, and it will be ignored. See Chapter 11, “Linker Commands”, for the correct syntax.

313 Assigning an address with '*' is not permitted (WARNING)

An asterisk symbol (*****) in the **ORDER** command cannot be assigned an address. See Chapter 11, “Linker Commands”,, for the correct syntax.

314 Cannot find section: name or section has already been ORDERed (WARNING)

The linker cannot locate this section. The operand has not been specified properly. This section reference will be ignored. See Chapter 11, “Linker Commands”, in this manual for more information on the **ORDER** command.

315 Maximum memory exceeded (ERROR)

An address assigned during the allocation stage has exceeded the range set for the target environment.

316 Entry point not found, starting address set to 0x0 (WARNING)

No entry point name was given on the command line, and the linker could not find the default entry point names.

317 Cannot open file: filename (ERROR)

The file does not exist or cannot be opened for reading.

318 No input object file (FATAL)

No object file names were provided on the command line in the absence of a command file or, if a command file is specified under the same condition, no **LOAD** commands were found. If you are attempting to load a library file as the input object file, the “force load” option (**-u**) must be provided at linker invocation to force an initial search of the library.

319 Cannot reassign value of PUBLIC symbol: symbol (WARNING)

The **PUBLIC** command cannot be used to reassign values to absolute symbols. The command will be ignored.

320 Illegal ORDER command; address assigned without section name (WARNING)

Only an address was specified but no section name was given for one of the **ORDER** command entries. The address will be ignored.

321 Section name required for SECTSIZE command (ERROR)

The **SECTSIZE** command requires a section as an argument.

322 Expecting a non-empty string as an argument (ERROR)

A quoted string is normally expected as an argument to this command. The string must also contain at least one character for the string to be valid. See Chapter 11, “Linker Commands”, for more information.

323 Cannot assign common symbol: name in PUBLIC command (WARNING)

The **PUBLIC** command has been used to reassign a value to a common symbol. The command will be ignored.

324 Section section deleted by linker, referenced elsewhere (FATAL)

A section is being referenced that does not exist within the linker any more. This error could indicate an internal error within the linker.

325 Illegal command for incremental linking - command ignored. (WARNING)

If incremental linking is specified, only the **INCLUDE**, **LISTMAP**, **LOAD**, and **SYMTRAN** commands can be used.

326 Internal error occurred: text (FATAL)

The linker experienced an internal error. The reason for the failure is indicated at the end of the message. Please report this inconsistency to Mentor Graphics Technical Support.

327 Invalid address address replaced by address address (WARNING)

The address you specified for a section cannot be properly aligned. The linker will automatically correct it to the alignment factor in effect for that section, if possible.

328 Invalid syntax (ERROR)

The command has not been properly specified. See Chapter 11, “Linker Commands”, for more information on this command.

329 Reserved memory overlapped from address address to address (WARNING)

An attempt has been made to overlap sections in memory.

330 Assigned entry point not found (ERROR)

You have given a symbol name as the entry point name and it cannot be found in the symbol table. The entry point will be set to zero.

331 Second '*' in ORDER command ignored (WARNING)

Only one asterisk (*****) is allowed in an **ORDER** command. See Chapter 11, “Linker Commands”, for the correct syntax.

332 SECTSIZE value for section section is less than actual section size (WARNING)

The **SECTSIZE** command is used to set the size of a section, which is normally smaller than the value used with the command. This warning indicates that the section's size is larger than the indicated value and the **SECTSIZE** command is ignored for that section.

333 Setting section size to value for section section (INFORMATIONAL)

This indicates that the size of the specified section was increased to the specified value. This occurs directly as a result of a **SECTSIZE** command.

335 Available Lilypond space exhausted: size (FATAL)

The size of the created lilypond was not sufficient for the number of lilypads that were needed. This application may require compilation with the **-KLf1** option. This is possibly an internal error. Please report this inconsistency to Mentor Graphics Technical Support.

336 Lilypond size exceeds 32M maximum size: size (FATAL)

If a lilypond is created that is over 32 Megabytes in size, the linker will not be able to place it in memory such that all of its contents can be accessed with a single absolute branch instruction. Thus, a valid lilypond cannot be created for this application. Applications should be compiled with **-KLf1** or higher.

337 Too many sections (FATAL)

The total number of sections exceeded the number that were allocated. This is an internal error. Please report this inconsistency to Mentor Graphics Technical Support.

338 Cannot process section: name of file: filename (WARNING)

The section cannot be found or has already been processed by a previous **ORDER** command. The section reference will be ignored.

339 Initializing section failed (ERROR)

The linker failed to set up an internal section list structure due to memory restrictions in the system.

340 Bad symbol value (ERROR)

The linker has discovered an internal inconsistency in the symbol value. Please report this inconsistency to Mentor Graphics Technical Support.

341 Unexpected information about .init section contents: file (ERROR)

An unrecognized **-KLi** setting was found in the indicated file. The linker does not know what the setting means.

342 Out of memory when buffering output file record (ERROR)

The linker has run out of space to store its internal data and has halted without completing its output. If you are linking with a library, use the librarian to remove modules that are never referenced.

343 Invalid flag option: string (WARNING)

The flag option is invalid for the command line. Fix the error by specifying a valid option. Valid options are listed in Chapter 2, “Command Usage”.

344 Invalid address/value: string (WARNING)

An address or value specified in the command has not been properly specified. The entry will be ignored.

345 Missing argument (WARNING)

The linker command is missing an argument. See Chapter 11, “Linker Commands”, for the correct syntax.

346 Unable to determine simulation requirements for final variant (WARNING)

The linker was unable to find information about the PowerPC variant that was determined as a result of the checking the object files that are being linked together. This error probably signals an internal error. Please report this inconsistency to Mentor Graphics Technical Support.

347 Unknown command (WARNING)

An unrecognizable command has been found in the command file. This command will be ignored.

348 Cannot process user-defined symbol: name (WARNING)

The linker is unable to create the external symbol that you defined in the command line. This symbol reference will be ignored.

349 Unknown relocation type at offset address in section: name of file: filename (ERROR)

Unknown relocation type at offset *value* has been encountered.

350 Problem with ABSOLUTE command, section: section ignored (WARNING)

The same section has been specified in more than one **ABSOLUTE** command. The section will still be output to the absolute file.

351 ALIGN ignored for string (WARNING)

The section specified by the **ALIGN** command is not found or the section cannot be aligned.

352 Section: name has been included by INITDATA already (WARNING)

Section *name* has been used more than once in an **INITDATA** command. The section will only be copied once into the special section created by the linker.

353 Input object contains incorrect information about .init section contents: file (ERROR)

The **-KLi** settings in the indicated object file is not complete. It is possible that this object file has been incrementally-linked with a non-Mentor Graphics tool.

354 Input objects contain incompatible .init sections: file (ERROR)

Objects with different and incompatible **-KLi** settings were linked together. All object modules that are linked together must have settings of either **-KLi0** or **KLi1**. The offending object files are indicated.

355 Input objects contain old-style initflag value of 2: file (WARNING)

An earlier version of the PowerPC C Compiler used the value of the **-KLf** command line option to determine how the contents of the .init section were to be used. This option was able to use the values of 0, 1, or 2. The compiler now uses the value of the **-KLi** command line option to determine how the **.init** section contents are to be used. That option only has values of 0 or 1. So, a value of 2 indicates that an older version of the C compiler was used to create that object file. While this value is allowed for now, it may be disallowed in the future.

356 Ambiguous ABSOLUTE command, conflicts with ORDER command (ERROR)

The section specified by the **ABSOLUTE** command is ambiguous to the linker because it conflicts with the **ORDER** command on the same section.

357 Invalid object file: filename (FATAL)

An input file has been found that is not a ELF relocatable object file.

358 Incremental link option overrides strip option (WARNING)

The strip option, **-s**, may only be used with absolute links. If an incremental link is performed, the strip option is ignored.

359 Invalid section type: name (WARNING)

The section type that you specified is not **text**, **data**, **lit**, or **bss**.

360 Cannot link absolute file: filename (FATAL)

The linker can link with an object file but not with an absolute file. Fix the error by specifying object files on the command line or in the command file.

362 Start address is an odd address (WARNING)

The entry point is at an odd address. The linker will continue but will diagnose this potentially troublesome situation.

363 Module compiled with incompatible option (option); using library for option (WARNING)

Input objects with potentially incompatible compilation options detected. This could cause problems in the resulting executable file.

364 Section: name has been reordered already (WARNING)

All the sections in the combined section have already been reordered. No further reordering can be specified.

365 File name- filename not allowed in command command (WARNING)

A file name is not a legal argument to the specified command. See Chapter 11, “Linker Commands”, for more information.

366 Section name and type missing (WARNING)

In the **ABSOLUTE** command, the section name or section type must be specified. See Chapter 11, “Linker Commands”, for the correct syntax.

367 Section, Type or Filename string too long, truncated (WARNING)

A section, type, or filename can have a maximum of 8192 characters. The name has been truncated.

368 Unresolved external: symbol_list (ERROR)

The symbols listed after this message have been referred to as externals but have not been defined as publics.

372 Cannot process section with type NOTE, DEBUG, RELOC, or OTHER: section (WARNING)

Debug, note, or relocatable sections cannot be manipulated with the linker commands. These sections are strictly for consumption by tools such as the linker or debugger and do not take up any space in the processor's memory image.

373 Entry point redefined (WARNING)

The entry point has been set more than once.

374 Bad Public (WARNING)

The value assigned to a symbol using the **PUBLIC** command is not a valid constant or relocatable expression. See Chapter 11, “Linker Commands”, for more information.

376 Can't mix little and big endian: module is big/little endian; previous objects have been little/big endian (FATAL)

The ELF format does not allow the mixing of big- and little-endian objects into a single resulting object. All object files being linked together must have the same endianity. One possible cause of this error message is using the wrong version of a library.

380 Too many (value) libraries to be linked, maximum number allowed: value (ERROR)

The linker has a limit to how many libraries it can keep track of. This error message is used to let the user know if this limit has been exceeded. If such an error occurs, fewer libraries must be used somehow, either by combining libraries or by doing incremental links.

382 No -c or -p option specified; using default command file filename (WARNING)

If the **no** command filename is provided using the **-c** option, or the processor variant is not provided using the **-p** option, the linker will select a default command file. The user may wish to verify whether this command file is adequate.

387 Small Data Area Out of Addressable Range from base_symbol (FATAL)

Small data area sections cannot be addressed from this base symbol using a 16-bit signed offset. If the base value is defined in the linker command file, reassign its value or use the **ORDER** command to rearrange the placement of the small data area sections. If a value is not specified, use the **ORDER** command to relocate the small data area sections as close together as possible.

388 Small Data Area Base: base_symbol Must be Absolute (FATAL)

The base symbol for small data area cannot be a relocatable symbol. Try providing the definition through a **PUBLIC** linker command.

389 Value Too Large for Displacement Field at Offset: address (ERROR)

The value computed for relocation at offset *address* is too big for the displacement field. The error might be caused by a wrong relocation type for this type of instruction.

390 Bad Displacement Value at Offset address for relocation type: reloc_type (ERROR)

The displacement value computed for this relocation type has an incorrect format. The error might be caused by a wrong relocation type for this type of instruction.

391 Invalid Relocation Entry at offset address of Type: reloc_type (ERROR)

The relocation entry is invalid for this type of instruction.

392 Relocation symbol: string is Not an SDA Symbol (ERROR)

The relocation symbol for this relocation entry must be a symbol defined in one of the Small Data Areas. Check the assembly source for a potentially bad operator on the symbol.

393 Bad Alignment of Relocation Storage Unit at Offset: address (ERROR)

The target storage unit for the relocation entry is not aligned on a proper boundary. Check assembly sources to see if the relocation is in a section of proper type.

394 Sizeof Symbol: string Must be Absolute (ERROR)

The **sizeof** symbol cannot be a relocatable symbol. Try defining the symbol using a **PUBLIC** linker command.

395 Value for section is Different Than Actual Section Size (WARNING)

The value supplied for this **sizeof** symbol is not the same as the actual size of the section. This can occur if more than one section exists with the same name but different types, or if a section was fragmented into multiple parts through the use of the **ORDER** directive. The linker is unable to determine an appropriate section size under these conditions.

396 Error While Creating .init Section (ERROR)

The linker failed to create an internal section for the initialization routines.

397 Cannot Find Symbol ._term_init for Initialization Routines (ERROR)

The linker could not update the initialization routines with the value of this symbol. Make sure all required libraries are included in the link process.

398 Initialization Routines Too Far From Startup Code (ERROR)

The symbol **._term_init** is located too far from the initialization routines. Try reordering sections to place the **.init** section closer to the section containing the **._term_init** symbol.

399 Bad symbol Index for Relocation Type reloc_type: string (ERROR)

The linker found an inconsistent symbol index in the relocation entry.

400 Multiple entries for section section found for sizeof() operator (WARNING)

The **sizeof()** operator is meant to retrieve the size of the section that was used with that operator. If there are multiple sections with the same name, which can occur through the use of the **ORDER** linker command, then the linker cannot determine which section’s size is to be used. This operator should not be used if multiple sections exist with a given name.

401 String instruction simulation library required

The output object file contains instructions that will not execute in hardware on the target processor it was built for. The linker has determined that the simulation code necessary to interpret those instructions is missing.

If an emulation routine is available, this warning can be avoided by defining the **._PPC_EMB_sim602_present** public symbol with a value of 0 as follows:

```
PUBLIC ._PPC_EMB_sim602_present=0
```

This may be appropriate if the emulation routine is already present in ROM.

You can also use the software emulation library provided by the MCCPPC compiler by uncommenting the following lines in the linker command file for that processor.

```
; EXTERN  ENT._FPE_main
; ORDER   SEC._FPE_main=0xFFF01000
```

402 Too many nested INCLUDE files (WARNING)

Include file can only be nested up to 16 deep. The command file should be reorganized to avoid this much inclusion of other files.

403 Path name of INCLUDE/command file is too long (WARNING)

The indicated include file or command file name is too long to be represented. The file should be moved to a different directory or renamed so as to avoid exceeding the 1024 character limit on UNIX hosts or the 260 character limit on Windows XP, Windows Vista, or Windows 7.

404 -g conflicts with -i, -g option is ignored (WARNING)

The **-g** option can only be used when an absolute file is created. If this option is used during an incremental link, it will be ignored.

405 Module contents incompatible with expected processor type (ERROR)

The contents of the object file that was just loaded is not compatible with the target processor type for the output object file. The resulting file may not execute appropriately. All input objects that are linked together must be either of the same processor type or be compatible with each other.

406 Processor word contains unsupported flags, flags removed: module (WARNING)

The processor information for the object file contains information that is not recognized by the linker. This information is removed since the linker does not know what the appropriate action would be. This warning may indicate a corrupt object or potentially an object created by a newer toolkit that is not understood by this version of the linker.

407 Module contains instructions incompatible with its processor type: module (WARNING)

The object file’s contents are not appropriate for its processor type. Use of this object file will result in an output file that may not execute properly.

408 Floating-point instruction simulation library required (ERROR)

The output object file contains instructions that will not execute in hardware on the target processor it was built for. The linker has determined that the simulation code necessary to interpret those instructions is missing.

If an emulation routine is available, this warning can be avoided by defining the **._PPC_EMB_sim_present** public symbol with a value of 0, as follows:

```
PUBLIC ._PPC_EMB_sim_present=0
```

This may be appropriate if the emulation routine is already present in ROM.

You can also use the software emulation library provided by the MCCPPC compiler by uncommenting the following lines in the linker command file for that processor.

```
; EXTERN  ENT._FPE_main
; ORDER   SEC._FPE_main=0xFFF01000
```

409 Module is missing processor type information: module (WARNING)

The indicated module does not contain information that represents the target processor that the file was created to execute on. Other modules that were linked with this module do contain this information. This warning indicates that linking with this module may produce an object or executable that may not execute appropriately on whatever target the other modules were targeted for. This message may also indicate that this module was produced by a non-Microtec toolkit or an older Microtec toolkit.

410 Module contains invalid processor word: module (ERROR)

The linker has detected that the indicated module contains invalid processor type information. This error may indicate a corrupt object or potentially an object created by a newer toolkit that is not understood by this version of the linker.

411 Start address outside available physical memory: value (ERROR)

The indicated start address cannot be addressed by the processor this executable was built to run on. The start address must be changed to fit within the memory space of the intended processor.

412 Can't open output file FILE for writing, since it has the same file name as an input file (FATAL)

The name of the indicated output file conflicts with an input file that has the same name. Since any attempt to write to the output file would destroy the contents of the input file, the linker will not open the file for output and stops execution.

413 Start address is not on 4-byte boundary (ERROR)

The address specified through the use of the **START** linker command or the -e entry point must be on a 4-byte boundary. This is required since the PowerPC processors can only execute instructions that are aligned to 4-byte boundaries.

414 Symbol: SYMBOL replacing definition from file: file (WARNING)

The indicated public symbol was defined in both an input object module and in the linker command file. The definition from the command file will be used by the linker.

415 no glue-code was used, recommend using -Zb0 (INFORMATIONAL)

The linker created lilyponds in case branches were used that required them. In this case, the application linked without requiring any such branches. As such, it might be worth relinking the application with the **-Zb0** option (or removing the existing **-Zb** option from the command line) in order to save some unneeded memory.

416 Invalid or missing information for processor variant: name (FATAL)

A problem was found when trying to get information for the indicated PowerPC variant. This error probably indicates an internal error. Please report this inconsistency to Mentor Graphics Technical Support.

801 ELF Internal Error

Invalid ELF information, such as file type, machine type, and so forth, encountered while writing an ELF object file.

802 ELF Write Error

File write error encountered while writing an ELF object file.

803 ELF out of memory

No memory was available to hold intermediate data while writing an ELF object module.

804 Invalid Symbol Binding

Invalid symbol binding value encountered while writing an ELF object file.

805 Invalid Symbol Type

Invalid symbol type encountered while writing an ELF object file.

806 Invalid Symbol Ordering

Some local symbols appear after global symbols in the ELF symbol table while writing to an ELF object file.

807 Invalid Relocation Type

Invalid relocation type encountered while writing an ELF object file.

808 Section Does Not Exist

Destination section does not exist while writing an ELF object file.

808 Section Already Exists

Duplicate section creation encountered while writing an ELF object file.

809 Name too long

The length of a symbol name longer than 8192 characters encountered while writing an ELF object file.

810 ELF Read Error

The linker could not read the input ELF object file. The input ELF object file may be corrupted.

811 Program header table overlaps ORDERed section

An invalid output object file was produced.

Appendix D
Librarian Error Messages

This appendix describes the error messages and warnings that appear if errors are detected while running the librarian. The error message is printed on the listing immediately following the statement in error.

Note

Messages that are outside the scope of this product (for example, operating system error messages) are not documented in this appendix.

If you encounter an undocumented error message, please contact Mentor Graphics Technical Support with the exact text of the message and the circumstances under which it occurred. The Technical Support representative will assist you in determining the cause of the problem.

Librarian Messages and Errors

100 **Could not close file *filename* to open another file**

In order to reduce processing overhead, the librarian keeps files open with the **OPEN** command. This message is displayed if too many files are open and the librarian unsuccessfully attempts to close a file in order to open a new one. To remedy this situation, reduce the number of files you are working with during a given session.

101 **Unable to open file *filename***

The librarian could not open the named file when executing an **ADDMOD**, **REPLACE**, or **OPEN** command. This error could be caused by either an invalid filename specification or when the specified file does not exist. The librarian ignores the command that generates this error.

102 **Unable to close file *filename***

The librarian generates this error when it encounters an operating system error and cannot close the named file. This message typically is accompanied by another error message that provides a more specific reason for not closing the named file.

103 **Unable to remove file *FILE***

There was a problem in unlinking the indicated file. This may indicate an internal error in the librarian.

104 **File *filename* not included**

The librarian issues this message when it cannot execute the **ADDMOD** command because the named file is corrupted or does not exist. This message has a companion message that specifically states why the named file is not included in the library.

106 File *library_name* exists already

The librarian generates this message when you use the **CREATE** command and the named library currently exists. The librarian displays a warning in batch and interactive modes.

107 File *filename* does not exist

The librarian generates this message when you issue an **OPEN** command and the named file does not exist.

108 Library file *library_name* not opened

This message has a companion message that specifically states why the library file was not opened.

109 Library file *library_name* not included

This message has a companion message that specifically states why the library file was not included.

120 Use HELP for proper command syntax

This message suggests using the **HELP** librarian command, which will display the correct syntax for all librarian commands that can be entered at this point in the session.

201 Module *module_name* not found

The librarian could not locate the named module in the library to execute a **DELETE**, **REPLACE**, or **EXTRACT** command.

203 Module *module_name* already exists in current library.

The librarian cannot execute an **ADDMOD** or **ADDLIB** command because the module named in the message exists in the current library. If you wish to replace a module in the library, use the **REPLACE** command. The librarian ignores the **ADDMOD** or **ADDLIB** command that contains a duplicate module name.

204 *filename* is a library file

The librarian generates this error message when it attempts to execute an **ADDMOD** command and the associated filename is not an object module. The command containing the erroneous file is ignored.

205 *filename* is not a library file

The librarian issues this command when it attempts to execute an **ADDLIB** or **OPEN** command and the associated file name is an object module. The librarian ignores the command containing the erroneous file.

206 Module *module_name* is not being included in the library

The librarian issues this message with a companion message that gives the specific reason for not including the named module in the library. This messages describes the result: the named module is not included in the library.

207 Bad object record

Either the object module has been corrupted or it is not a legal relocatable object file. The librarian issues this message with a companion message, which names the file with the bad object record. Whatever command is associated with the bad object record file will be ignored.

208 Bad library header record

The library has a bad header record. The librarian issues this message with a companion message, which names the file with the bad header record. The command associated with the bad library header record will be ignored.

209 Duplicate symbol *symbol_name*

A module named in an **ADDLIB**, **ADDMOD**, or **REPLACE** command has the same public definition symbol that occurs in another module. The librarian issues this message with a companion message that provides information about what action it takes.

The librarian considers symbols to be case-sensitive.

210 Bad object record in file *filename*

The named library or module file may have been corrupted.

211 Module *module* added

An informational message indicating that the specified module was added to the current library.

250 Out of memory

The librarian issues this message when it encounters insufficient system memory to execute commands issued since the last **CREATE** or **OPEN** command.

251 Failed writing library. *Reason*

The librarian generates this message when it attempts to execute a **SAVE** command, and cannot. The message provides the reason for the inability to create a library. The librarian abandons the current session affected by the **SAVE** command that caused the error.

252 fseek or ftell error

The librarian issues this message when either of the two system calls **ftell** or **fseek** fail.

253 Library *library_name* not written

The librarian issues this message when an error that occurs earlier in the session prevents the library from being saved. This message is typically accompanied by another message that contains the reason the named library was not created.

254 Failed writing module *module_name* to file *filename*

When attempting to execute an **EXTRACT** command, the librarian cannot write the named module from an existing library to the new file that is external to the library. A companion error message describes the reason that the module cannot be extracted. If an error is encountered in batch mode, all commands following the **EXTRACT** command will not be executed; however, they will still be checked for syntactical validity.

255 Replacement not done

The librarian issues this message when it cannot execute the **REPLACE** command for the reason specified in the companion message.

256 Extraction failed

The module named in the **EXTRACT** command is not extracted.

257 Illegal command

The librarian generates this message when it encounters an incorrect command sequence or an incorrect command syntax. For example, if your batch file does not have a terminating **END** or **QUIT** command, this error is generated.

258 Abrupt ending of comment

A **NULL** character was encountered before the **end-of-line** in a comment.

259 Quote not terminated

The closing quote on a quoted string could not be found.

262 There is no library to be saved

The librarian generates this message when the user issues a **SAVE** command without first creating a library.

263 *Filename* contains an unrecognized or corrupted module

The file given to the **ADDMOD** command is not a legal module.

265 Symbol *symbol* has been truncated

The indicated symbol exceeded the allowable symbol lengths for the librarian. The displayed symbol has been truncated.

266 Demangled name of Symbol *symbol* has been truncated

The indicated symbol exceeded the allowable symbol lengths for the librarian. The displayed symbol has been truncated.

267 Library *library* contains no modules.

The librarian was unable to process an **EXTRACTALL** command because the current library contains no modules. No other actions were taken as a result of this command.

268 Input line too long: text truncated

An input line was processed that contained more than 8200 characters. Any characters beyond that limit were lost. The command should be broken up onto multiple lines in order for the librarian to be able to process it.

269 Module *module* already exists in current library

The librarian cannot execute an **ADDMOD** or **ADDLIB** command because the module named in the message exists in the current library. If you wish to replace a module in the library, use the **REPLACE** command. The librarian ignores the **ADDMOD** or **ADDLIB** command that contains a duplicate module name.

270 Library *library* not written

The librarian issues this message when an error that occurs earlier in the session prevents the library from being saved. This message is typically accompanied by another message that contains the reason the named library was not created.

271 Replacement not done

The librarian issues this message when it cannot execute the **REPLACE** command for the reason specified in the companion message.

272 File *file* not included

The librarian issues this message when it cannot execute the **ADDMOD** command because the named file is corrupted or does not exist. This message has a companion message that specifically states why the named file is not included in the library.

273 Module *module* not found

The librarian could not locate the named module in the library to execute a **DELETE**, **REPLACE**, or **EXTRACT** command.

Appendix E
C++ Support

This appendix discusses modifications to the ASMPPC Assembler, LNKPPC Linker, and LIBPPC Librarian to support the C++ language. This support affects both DWARF debug information and C++ symbol name mangling and demangling.

DWARF Debug Information

The standard file format used by the Microtec ASMPPC toolkit is ELF. Debug information for C and C++ modules are stored in the DWARF 2 format in the special debug sections **.debug_abbrev**, **.debug_aranges**, **.debug_frame**, **.debug_info**, **.debug_line, .debug_loc**, **.debug_macinfo**, **.debug_pubnames**, and **.debug_str**.

Name Mangling and Demangling

In the C++ language, you can use the same names to refer to different operations; this practice is known as “overloading.” Since the same name is used, C++ introduced a unique naming encoding scheme to differentiate between two names used for the different operations. Therefore, you can name one function:

```
closeout(char *name, float balance)
```

to indicate closing out a bank account, pass it a parameter indicating the customer’s name, and return a parameter indicating the closing balance. You can name another function in your program:

```
closeout(char *telex, int *account)
```

indicating a bank branch to notify about the customer’s closed account. Although both these functions have the same name (**closeout**), C++ uniquely identifies each function based on the parameters passed by encoding this information into a “mangled” name.

ASMPPC Assembler

A sample C++ file, **simple1.cc**, was compiled on a UNIX operating system. The resulting **simple1.o** object file was linked with the map option set so that a link map was generated. Figure E-1 shows the sample C++ source file.

Figure E-1. C++ Program Listing

C++ file:

`simple1.cc`

```
#include <iostream.h>
int main() {
   cout << "hello";
}
```

Use the **.lflags** directive in your assembler source file to view all labels that fit the C++ name encoding scheme in your cross-reference and symbol tables. Both the mangled and demangled C++ names (the original C++ source file name) will be listed for the cross-reference and symbol tables in the assembler output listing. The following example shows a portion of the symbol table from the assembler output file generated for **simple1.cc**:

Example E-1. Generated Symbol Table

```
         Symbol Table

Label             Value

_IosInit          .data :00000001
__ct__41basic_ios__tm__24_c20char_traits__tm__2_cFv
basic_ios<T1,T2>::_ct() [with T1=char, T2=char-Traits<char>]
                  .text :00000C2C
__dt__41basic_ios__tm__24_c20char_Traits__tm__2_cFv
basic_ios<T1,T2>::_dt() [with T1=char, T2=char-Traits<char>]
                  .text :00000680
__record_needed_destructionExternal
__vtbl__41basic_ios__tm__24_c20char_traits__tm__2_c
basic_ios<T1,T2>::_vtbl [with T1=char, T2=char-Traits<char>]
                  .rodata :00000030
_main             External
main              .text :000005F8
memmove           External
```

LNKPPC Linker

The LNKPPC Linker also supports C++ name mangling and demangling. Since the mangled name is not easily interpreted, the decoded (or demangled) name is also shown on the linker error message output and map files. In the map file output, C++ mangled names can appear in the **GLOBAL SYMBOL TABLE** section. If a global linkage name fits the C++ name mangling pattern, LNKPPC will report the C++ mangled name as well as the C++ demangled name in the map file.

Example E-2. Mangled Names in the Symbol Table

One encoded name of a function defined in the C++ I/O stream class library is:

```
_M_open__13_Filebuf_baseFPCci
```

LNKPPC will decode the above encoded C++ name into the C++ symbolic source name:

```
    _Filebuf_base::M_open(const char *, int)
```

Now, you can see that the class **_Filebuf_base** contains a function **M_open**, which takes two arguments, one that is a **constant char *** and another one that is an integer.

LNKPPC will report the mangled name and the demangled name in the global symbol table entry of that function in the linker map file:

```
GLOBAL SYMBOL TABLE
-------------------

SYMBOL   SECTION  VALUE
      ....
_Filebuf_base::M_open(const char *, int)
_M_open__13_filebuf_baseFPCci
         .text    00024900
      ....
```

Symbol Name Length

LNKPPC's linkage name limit is 8192 characters, which allows for longer mangled names. For long symbol names, LNKPPC lists the symbol on a line by itself. The **SECTION**, **ADDRESS**, and **MODULE** information will start in the corresponding columns of the next line following the long symbol name.

A link map was generated for **simple1.cc** running on a UNIX-based host. The following list describes several C++ unique items that appear in the link map:

- The **.init** section is a special C++ section reserved for pointers to C++ static constructors and destructors. For more information, refer to your C++ Compiler reference manual:

```
SECTION SUMMARY
---------------
SECTION  TYPE  PROGSEG  START    END      SIZE        ALIGN    Module
.init    TEXT  @TEXT    000100C8 000100FF 00000038    4        simple1.o
```

- Mangled names have been decoded to show their demangled names. Symbolic names are listed in mangled or demangled pairs:

```
      _Filebuf_base::M_open(const char*, int)
      _M_open__13_Filebuf_baseFPCci
```

- Names prefixed by **__vtbl__** are the linkage names for pointers to the C++ virtual tables produced by the C++ compiler:

```
      runtime_ereror::_vtbl
      __vtbl__13runtime_error
```

LIBPPC Librarian

The LIBPPC Librarian will automatically demangle C++ names when they are encountered if a given symbol name fits the C++ name mangling scheme. If there is a C++ symbolic source-level name available for the symbol name being processed, LIBPPC will display the C++ symbolic name in the next line of the low-level name:

```
<low-level mangled c++ name>
C++ NAME:  <c++ demangled symbolic source-level name>
```

Example E-3. Symbolic Name Display with LIBPPC

```
_M_open__13_Filebuf_baseFiT1
C++ NAME:_Filebuf_base::M_open(int, int)
```

In this example, the mangled name shows the function signature. The demangled name shows that there is a function **M_open** in the **_Filebuf_base** class that takes two **int** arguments.

Glossary

Address Expression

An expression whose value represents a location in memory.

Base Address

Lowest address considered for loading relocatable sections of the absolute object module.

Load Address

Memory address where the lowest byte of a section is placed.

Module

The relocatable object code resulting from a single assembly. It can contain pieces of one or more sections.

The linker combines pieces of a section from different modules. Such pieces always make up a contiguous block of memory, assuming they can be combined at all.

Numeric Expression

An expression whose value represents a number.

Register Expression

An address expression whose base or index attribute is not null.

Relocatable Section

A general purpose section that can contain both instructions and/or data.

Starting Address

The location where execution begins.

Subsection

Individual pieces of code from various modules that make up a section.

Index

Embedded Software and Hardware License Agreement

The latest version of the Embedded Software and Hardware License Agreement is available on-line at:
www.mentor.com/eshla

IMPORTANT INFORMATION

USE OF ALL PRODUCTS IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF PRODUCTS INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

EMBEDDED SOFTWARE AND HARDWARE LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Products (as defined in Section 1) between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Products received electronically, certify destruction of Products and all accompanying items within five days after receipt of such Products and receive a full refund of any license fee paid.

1. **Definitions.** As used in this Agreement and any applicable quotation, supplement, attachment and/or addendum ("Addenda"), these terms shall have the following meanings:

1.1. "Customer's Product" means Customer's end-user product identified by a unique SKU (including any Related SKUs) in an applicable Addenda that is developed, manufactured, branded and shipped solely by Customer or an authorized manufacturer or subcontractor on behalf of Customer to end-users or consumers;

1.2. "Developer" means a unique user, as identified by a unique user identification number, with access to Embedded Software at an authorized Development Location. A unique user is an individual who works directly with the embedded software in source code form, or creates, modifies or compiles software that ultimately links to the Embedded Software in Object Code form and is embedded into Customer's Product at the point of manufacture;

1.3. "Development Location" means the location where Products may be used as authorized in the applicable Addenda;

1.4. "Development Tools" means the software that may be used by Customer for building, editing, compiling, debugging or prototyping Customer's Product;

1.5. "Embedded Software" means Software that is embeddable;

1.6. "End-User" means Customer's customer;

1.7. "Executable Code" means a compiled program translated into a machine-readable format that can be loaded into memory and run by a certain processor;

1.8. "Hardware" means a physically tangible electro-mechanical system or sub-system and associated documentation;

1.9. "Linkable Object Code" or "Object Code" means linkable code resulting from the translation, processing, or compiling of Source Code by a computer into machine-readable format;

1.10. "Mentor Embedded Linux" or "MEL" means Mentor Graphics' tools, source code, and recipes for building Linux systems;

1.11. "Open Source Software" or "OSS" means software subject to an open source license which requires as a condition for redistribution of such software, including modifications thereto, that the: (i) redistribution be in source code form or be made available in source code form; (ii) redistributed software be licensed to allow the making of derivative works; or (iii) redistribution be at no charge;

1.12. "Processor" means the specific microprocessor to be used with Software and implemented in Customer's Product;

1.13. "Products" means Software, Term-Licensed Products and/or Hardware;

1.14. "Proprietary Components" means the components of the Products that are owned and/or licensed by Mentor Graphics and are not subject to an Open Source Software license, as more fully set forth in the product documentation provided with the Products;

1.15. “Redistributable Components” means those components that are intended to be incorporated or linked into Customer’s Linkable Object Code developed with the Software, as more fully set forth in the documentation provided with the Products;

1.16. “Related SKU” means two or more Customer Products identified by logically-related SKUs, where there is no difference or change in the electrical hardware or software content between such Customer Products;

1.17. “Software” means software programs, Embedded Software and/or Development Tools, including any updates, modifications, revisions, copies, documentation and design data that are licensed under this Agreement;

1.18. “Source Code” means software in a form in which the program logic is readily understandable by a human being;

1.19. “Sourcery CodeBench Software” means Mentor Graphics’ Development Tool for C/C++ embedded application development;

1.20. “Sourcery VSIPL++” is Software providing C++ classes and functions for writing embedded signal processing applications designed to run on one or more processors;

1.21. “Stock Keeping Unit” or “SKU” is a unique number or code used to identify each distinct product, item or service available for purchase;

1.22. “Subsidiary” means any corporation more than 50% owned by Customer, excluding Mentor Graphics competitors. Customer agrees to fulfill the obligations of such Subsidiary in the event of default. To the extent Mentor Graphics authorizes any Subsidiary’s use of Products under this Agreement, Customer agrees to ensure such Subsidiary’s compliance with the terms of this Agreement and will be liable for any breach by a Subsidiary; and

1.23. “Term-Licensed Products” means Products licensed to Customer for a limited time period (“Term”).

2. **Orders, Fees and Payment.**

2.1. To the extent Customer (or if agreed by Mentor Graphics, Customer’s appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement (“Order(s)”), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement and any applicable Addenda, whether or not these documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order will not be effective unless agreed in writing by an authorized representative of Customer and Mentor Graphics.

2.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. All invoices will be sent electronically to Customer on the date stated on the invoice unless otherwise specified in an Addendum. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice(s). Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax, consumption tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer’s sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer’s behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.

2.3. All Products are delivered FCA factory (Incoterms 2010), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics’ delivery of Software by electronic means is subject to Customer’s provision of both a primary and an alternate e-mail address.

3. **Grant of License.**

3.1. The Products installed, downloaded, or otherwise acquired by Customer under this Agreement constitute or contain copyrighted, trade secret, proprietary and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software as described in the applicable Addenda. The limited licenses granted under the applicable Addenda shall continue until the expiration date of Term-Licensed Products or termination in accordance with Section 12 below, whichever occurs first. **Mentor Graphics does NOT grant Customer any right to (a) sublicense or (b) use Software beyond the scope of this Section without first signing a separate agreement or Addenda with Mentor Graphics for such purpose.**

3.2. License Type. The license type shall be identified in the applicable Addenda.

3.2.1. Development License: During the Term, if any, Customer may modify, compile, assemble and convert the applicable Embedded Software Source Code into Linkable Object Code and/or Executable Code form by the number of Developers specified, for the Processor(s), Customer’s Product(s) and at the Development Location(s) identified in the applicable Addenda.

3.2.2. End-User Product License: During the Term, if any, and unless otherwise specified in the applicable Addenda, Customer may incorporate or embed an Executable Code version of the Embedded Software into the specified number of copies of Customer's Product(s), using the Processor Unit(s), and at the Development Location(s) identified in the applicable Addenda. Customer may manufacture, brand and distribute such Customer's Product(s) worldwide to its End-Users.

3.2.3. Internal Tool License: During the Term, if any, Customer may use the Development Tools solely: (a) for internal business purposes and (b) on the specified number of computer work stations and sites. Development Tools are licensed on a per-seat or floating basis, as specified in the applicable Addenda, and shall not be distributed to others or delivered in Customer's Product(s) unless specifically authorized in an applicable Addenda.

3.2.4. Sourcery CodeBench Professional Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software (i) if the license is a node-locked license, by a single user who uses the Software on up to two machines provided that only one copy of the Software is in use at any one time, or (ii) if the license is a floating license, by the authorized number of concurrent users on one or more machines provided that only the authorized number of copies of the Software are in use at any one time, and (b) distribute the Redistributable Components of the Software in Executable Code form only and only as part of Customer's Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.

3.2.5. Sourcery CodeBench Standard Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software by a single user who uses the Software on up to two machines provided that only one copy of the Software is in use at any one time, and (b) distribute the Redistributable Component(s) of the Software in Executable Code form only and only as part of Customer's Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.

3.2.6. Sourcery CodeBench Personal Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software by a single user who uses the Software on one machine, and (b) distribute the Redistributable Component(s) of the Software in Executable Code form only and only as part of Customer Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.

3.2.7. Sourcery CodeBench Academic Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software for non-commercial, academic purposes only by a single user who uses the Software on one machine, and (b) distribute the Redistributable Component(s) of the Software in Executable Code form only and only as part of Customer Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.

3.3. Mentor Graphics may from time to time, in its sole discretion, lend Products to Customer. For each loan, Mentor Graphics will identify in writing the quantity and description of Software loaned, the authorized location and the Term of the loan. Mentor Graphics will grant to Customer a temporary license to use the loaned Software solely for Customer's internal evaluation in a non-production environment. Customer shall return to Mentor Graphics or delete and destroy loaned Software on or before the expiration of the loan Term. Customer will sign a certification of such deletion or destruction if requested by Mentor Graphics.

4. **Beta Code.**

4.1. Portions or all of certain Products may contain code for experimental testing and evaluation ("Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and Customer's use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form.

4.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer's use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer's evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.

4.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer's feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 4.3 shall survive termination of this Agreement.

5. **Restrictions on Use.**

5.1. Customer may copy Software only as reasonably necessary to support the authorized use, including archival and backup purposes. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Except where embedded in Executable Code form in Customer's Product, Customer shall maintain a record of the number and location of all copies of Software, including copies merged with other software and products, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer's employees, authorized manufacturers or authorized contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use Products except as permitted by this Agreement. Customer shall give Mentor Graphics immediate written notice of any unauthorized disclosure or use of the Products as soon as Customer learns or becomes aware of such unauthorized disclosure or use.

5.2. Customer acknowledges that the Products provided hereunder may contain Source Code which is proprietary and its confidentiality is of the highest importance and value to Mentor Graphics. Customer acknowledges that Mentor Graphics may be seriously harmed if such Source Code is disclosed in violation of this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive any Source Code from Products that are not provided in Source Code form. Except as embedded in Executable Code in Customer's Product and distributed in the ordinary course of business, in no event shall Customer provide Products to Mentor Graphics competitors. Log files, data files, rule files and script files generated by or for the Software (collectively "Files") constitute and/or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Under no circumstances shall Customer use Products or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Products, or disclose to any third party the results of, or information pertaining to, any benchmark.

5.3. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense or otherwise transfer the Products, whether by operation of law or otherwise ("Attempted Transfer"), without Mentor Graphics' prior written consent, which shall not be unreasonably withheld, and payment of Mentor Graphics' then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer's permitted successors in interest and assigns.

5.4. Notwithstanding any provision in an OSS license agreement applicable to a component of the Sourcery CodeBench Software that permits the redistribution of such component to a third party in Source Code or binary form, Customer may not use any Mentor Graphics trademark, whether registered or unregistered, in connection with such distribution, and may not recompile the Open Source Software components with the --with-pkgversion or --with-bugurl configuration options that embed Mentor Graphics' trademarks in the resulting binary.

5.5. The provisions of this Section 5 shall survive the termination of this Agreement.

6. **Support Services.**

6.1. Except as described in Sections 6.2, 6.3 and 6.4 below, and unless otherwise specified in any applicable Addenda to this Agreement, to the extent Customer purchases support services, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased in accordance with Mentor Graphics' then-current End-User Software Support Terms located at http://supportnet.mentor.com/about/legal/.

6.2. To the extent Customer purchases support services for Sourcery CodeBench Software, support will be provided solely in accordance with the provisions of this Section 6.2. Mentor Graphics shall provide updates and technical support to Customer as described herein only on the condition that Customer uses the Executable Code form of the Sourcery CodeBench Software for internal use only and/or distributes the Redistributable Components in Executable Code form only (except as provided in a separate redistribution agreement with Mentor Graphics or as required by the applicable Open Source license). Any other distribution by Customer of the Sourcery CodeBench Software (or any component thereof) in any form, including distribution permitted by the applicable Open Source license, shall automatically terminate any remaining support term. Subject to the foregoing and the payment of support fees, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased in accordance with Mentor Graphics' then-current Sourcery CodeBench Software Support Terms located at http://www.mentor.com/codebench-support-legal.

6.3. To the extent Customer purchases support services for Sourcery VSIPL++, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased solely in accordance with Mentor Graphics' then-current Sourcery VSIPL++ Support Terms located at http://www.mentor.com/vsipl-support-legal.

6.4. To the extent Customer purchases support services for Mentor Embedded Linux, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased solely in accordance with Mentor Graphics' then-current Mentor Embedded Linux Support Terms located at http://www.mentor.com/mel-support-legal.

7. **Third Party and Open Source Software.** Products may contain Open Source Software or code distributed under a proprietary third party license agreement. Please see applicable Products documentation, including but not limited to license notice files, header files or source code for further details. Please see the applicable Open Source Software license(s) for additional rights and obligations governing your use and distribution of Open Source Software. Customer agrees that it shall not subject any Product provided by Mentor Graphics under this Agreement to any Open Source Software license that does not otherwise apply to such Product. In the event of conflict between the terms of this Agreement, any Addenda and an applicable OSS or proprietary third party agreement, the OSS or proprietary third party agreement will control solely with respect to the OSS or proprietary third party software component. The provisions of this Section 7 shall survive the termination of this Agreement.

8. **Limited Warranty.**

   8.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual and/or specification. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Products under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification or improper installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY, PROVIDED CUSTOMER HAS OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; OR (B) PRODUCTS PROVIDED AT NO CHARGE, WHICH ARE PROVIDED "AS IS" UNLESS OTHERWISE AGREED IN WRITING.

   8.2. THE WARRANTIES SET FORTH IN THIS SECTION 8 ARE EXCLUSIVE TO CUSTOMER AND DO NOT APPLY TO ANY END-USER. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, WITH RESPECT TO PRODUCTS OR OTHER MATERIAL PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

9. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, AND EXCEPT FOR EITHER PARTY'S BREACH OF ITS CONFIDENTIALITY OBLIGATIONS, CUSTOMER'S BREACH OF LICENSING TERMS OR CUSTOMER'S OBLIGATIONS UNDER SECTION 10, IN NO EVENT SHALL: (A) EITHER PARTY OR ITS RESPECTIVE LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF SUCH PARTY OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES; AND (B) EITHER PARTY OR ITS RESPECTIVE LICENSORS' LIABILITY UNDER THIS AGREEMENT, INCLUDING, FOR THE AVOIDANCE OF DOUBT, LIABILITY FOR ATTORNEYS' FEES OR COSTS, EXCEED THE GREATER OF THE FEES PAID OR OWING TO MENTOR GRAPHICS FOR THE PRODUCT OR SERVICE GIVING RISE TO THE CLAIM OR $500,000 (FIVE HUNDRED THOUSAND U.S. DOLLARS). NOTWITHSTANDING THE FOREGOING, IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 9 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

10. **Hazardous Applications.**

    10.1. Customer agrees that Mentor Graphics has no control over Customer's testing or the specific applications and use that Customer will make of Products. Mentor Graphics Products are not specifically designed for use in the operation of nuclear facilities, aircraft navigation or communications systems, air traffic control, life support systems, medical devices or other applications in which the failure of Mentor Graphics Products could lead to death, personal injury, or severe physical or environmental damage ("Hazardous Applications").

    10.2. CUSTOMER ACKNOWLEDGES IT IS SOLELY RESPONSIBLE FOR TESTING PRODUCTS USED IN HAZARDOUS APPLICATIONS AND SHALL BE SOLELY LIABLE FOR ANY DAMAGES RESULTING FROM SUCH USE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF PRODUCTS IN ANY HAZARDOUS APPLICATIONS.

    10.3. CUSTOMER AGREES TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE OR LIABILITY, INCLUDING REASONABLE ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH THE USE OF PRODUCTS AS DESCRIBED IN SECTION 10.1.

    10.4. THE PROVISIONS OF THIS SECTION 10 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

11. **Infringement.**

11.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay any costs and damages finally awarded against Customer that are attributable to the action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.

11.2. If a claim is made under Subsection 11.1 Mentor Graphics may, at its option and expense, and in addition to its obligations under Section 11.1, either (a) replace or modify the Product so that it becomes noninfringing; or (b) procure for Customer the right to continue using the Product. If Mentor Graphics determines that neither of those alternatives is financially practical or otherwise reasonably available, Mentor Graphics may require the return of the Product and refund to Customer any purchase price or license fee(s) paid.

11.3. Mentor Graphics has no liability to Customer if the claim is based upon: (a) the combination of the Product with any product not furnished by Mentor Graphics, where the Product itself is not infringing; (b) the modification of the Product other than by Mentor Graphics or as directed by Mentor Graphics, where the unmodified Product would not infringe; (c) the use of the infringing Product when Mentor Graphics has provided Customer with a current unaltered release of a non-infringing Product of substantially similar functionality in accordance with Subsection 11.2(a); (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells, where the Product itself is not infringing; (f) any Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; (h) Open Source Software, except to the extent that the infringement is directly caused by Mentor Graphics' modifications to such Open Source Software; or (i) infringement by Customer that is deemed willful. In the case of (i), Customer shall reimburse Mentor Graphics for its reasonable attorneys' fees and other costs related to the action.

11.4. THIS SECTION 11 IS SUBJECT TO SECTION 9 ABOVE AND STATES: (A) THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS AND (B) CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.

12. **Termination and Effect of Termination.** If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized Term.

12.1. Termination for Breach. This Agreement shall remain in effect until terminated in accordance with its terms. Mentor Graphics may terminate this Agreement and/or any licenses granted under this Agreement, and Customer will immediately discontinue use and distribution of Products, if Customer (a) commits any material breach of any provision of this Agreement and fails to cure such breach upon 30-days prior written notice; or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination. For the avoidance of doubt, nothing in this Section 12 shall be construed to prevent Mentor Graphics from seeking immediate injunctive relief in the event of any threatened or actual breach of Customer's obligations hereunder.

12.2. Effect of Termination. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination or expiration of the Term, Customer will discontinue use and/or distribution of Products, and shall return Hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form, except to the extent an Open Source Software license conflicts with this Section 12.2 and permits Customer's continued use of any Open Source Software portion or component of a Product. Upon termination for Customer's breach, an End-User may continue its use and/or distribution of Customer's Product so long as: (a) the End-User was licensed according to the terms of this Agreement, if applicable to such End-User, and (b) such End-User is not in breach of its agreement, if applicable, nor a party to Customer's breach.

13. **Export.** The Products provided hereunder are subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct or indirect products thereof, to certain countries and certain persons. Customer agrees that it will not export Products in any manner without first obtaining all necessary approval from appropriate local and United States government agencies. Customer acknowledges that the regulation of product export is in continuous modification by local governments and/or the United States Congress and administrative agencies. Customer agrees to complete all documents and to meet all requirements arising out of such modifications.

14. **U.S. Government License Rights.** Software was developed entirely at private expense. All Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to US FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in this Agreement, except for provisions which are contrary to applicable mandatory federal laws.

15. **Third Party Beneficiary.** For any Products licensed under this Agreement and provided by Customer to End-Users, Mentor Graphics or the applicable licensor is a third party beneficiary of the agreement between Customer and End-User. Mentor

Graphics Corporation, Mentor Graphics (Ireland) Limited, and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.

16. **Review of License Usage.** Customer will monitor the access to and use of Software. With prior written notice, during Customer's normal business hours, and no more frequently than once per calendar year, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system, records, accounts and sublicensing documents deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FlexNet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all Customer information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. Such license review shall be at Mentor Graphics' expense unless it reveals a material underpayment of fees of five percent or more, in which case Customer shall reimburse Mentor Graphics for the costs of such license review. Customer shall promptly pay any such fees. If the license review reveals that Customer has made an overpayment, Mentor Graphics has the option to either provide the Customer with a refund or credit the amount overpaid to Customer's next payment. The provisions of this Section 16 shall survive the termination of this Agreement.

17. **Controlling Law, Jurisdiction and Dispute Resolution.** This Agreement shall be governed by and construed under the laws of the State of California, USA, excluding choice of law rules. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the state and federal courts of California, USA. Nothing in this section shall restrict Mentor Graphics' right to bring an action (including for example a motion for injunctive relief) against Customer or its Subsidiary in the jurisdiction where Customer's or its Subsidiary's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.

18. **Severability.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.

19. **Miscellaneous.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements, including but not limited to any purchase order terms and conditions. This Agreement may only be modified in writing, signed by an authorized representative of each party. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.

Rev. 120305, Part No. 252061