TEXAS INSTRUMENTS

TMS320C80 (MVP) Code Generation Tools

User's Guide

1995 Microprocessor Development Systems

TEXAS INSTRUMENTS

Printed in U.S.A., March 1995
SDS

SPRU108A

User's Guide

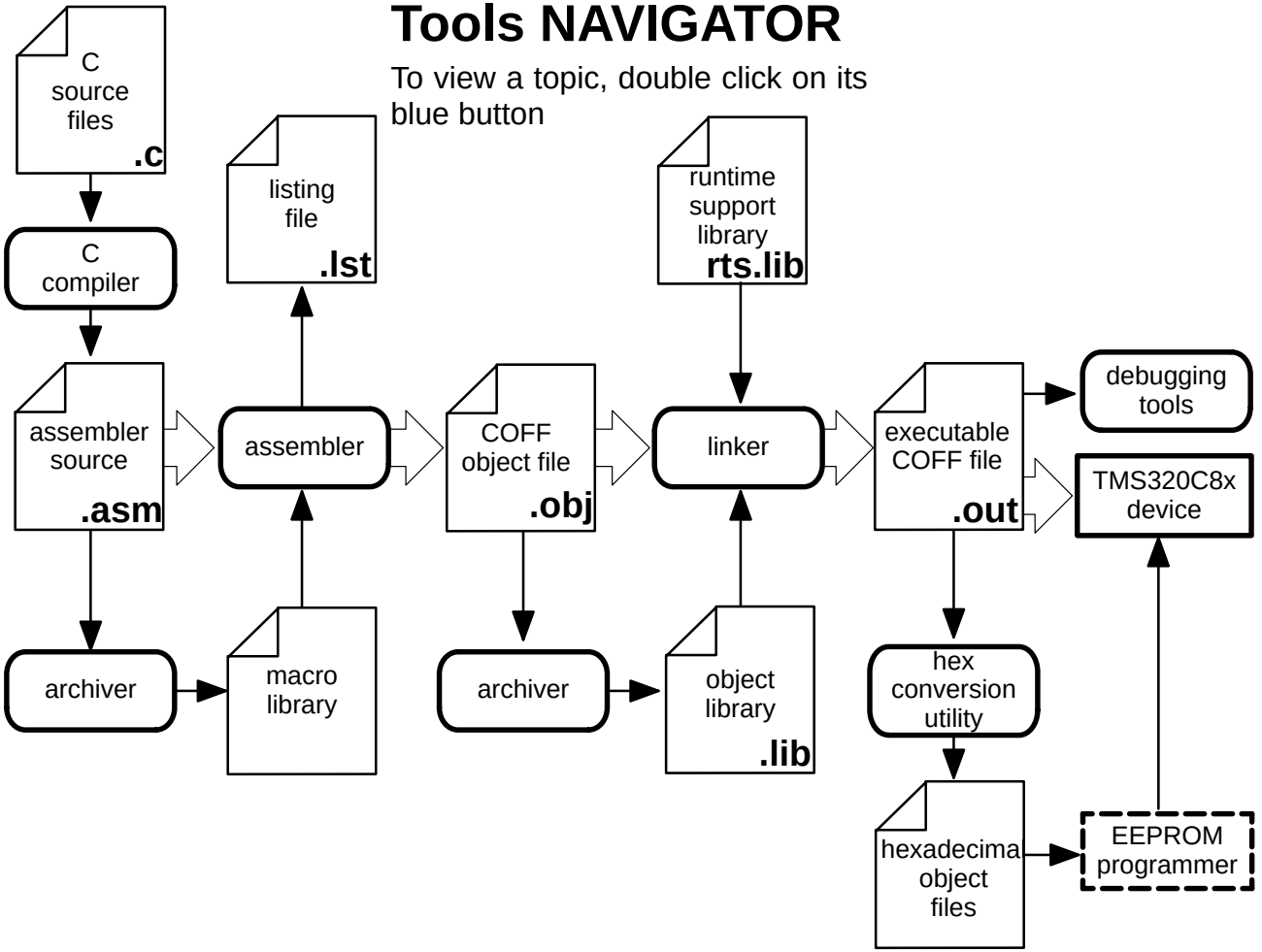
TMS320C80 (MVP) Code Generation Tools

1995

TMS320C80 (MVP) Code Generation Tools User's Guide

Code Generation Tools NAVIGATOR

To view a topic, double click on its blue button



The hex-conversion utility provides the ability to convert common object file format (COFF) files into a format that can be downloaded to EEPROM by an EEPROM programmer, however, an EEPROM programmer is not provided as part of the TMS320C8x Code Generation Tools.

The TMS320C8x Code Generation Tools User's Guide does not include a C programming reference. For that, we recommend Brian W. Kernighan and Dennis M. Ritchie's *The C Programming Language, Second Edition*, published by Prentice Hall.

For more information on the TMS320C8x device, please refer to the *System Level Synopsis* or one of the Processor/Hardware Documents available in the Bookshelf.

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Read This First

About This Manual

The TMS320C80 MVP (multimedia video processor) is Texas Instruments first single-chip multiprocessor DSP (digital signal processor) device. The MVP contains five powerful, fully programmable processors: a master processor (MP) and four parallel processors (PPs). The MP is a 32-bit RISC (reduced instruction set computer) with an integral, high-performance IEEE-754 floating-point unit. Each PP is an advanced 32-bit DSP; thus, in addition to having similar processing capabilities as conventional DSPs, each PP has advanced features to accelerate operation on a variety of data types.

The MVP supports a variety of parallel-processing configurations, which facilitates a wide range of multimedia and other applications that require high processing speeds. Applications include image processing, two- and three-dimensional and virtual reality graphics, audio/video digital compression, and telecommunications.

This manual describes the MVP code generation tools. The code generation tools package contains special shell utilities to compile, assemble, and link source files to create an executable object file with one command. The C compilers include an optimizer for producing highly optimized code. This manual provides information about the features and operation of the linker, and the MP and PP C compilers and assemblers. It also includes a description of the common object file format (COFF) and shows you how to link MP and PP code.

Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a `special typeface` similar to a typewriter's. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.). Here is a sample function call:

```
#include <headers.h>
main()
{
   call ();
}
```

- In syntax descriptions, the instruction, command, or directive is in a **bold face font**, and parameters are in *italics*. Portions of a syntax that are in **bold face** should be entered as shown; portions of a syntax that are in *italics* describe the type of information that should be entered. Syntax that will be entered on a command line is centered in a bounded box. Syntax that will be used in a text file is left-justified in an unbounded box. Here is an example of a directive syntax:

  #include "*filename*"

  The **#include** preprocessor directive has one required parameter, *filename*. The file name must be enclosed in quote marks.

- Square brackets (**[** and **]**) identify optional parameter. If you use an optional parameter, you specify the information within the brackets; you don't enter the brackets themselves. Here is an example of a command that has an optional parameter:

  mvphex [*–options*] *filename*

  The **mvphex** command has two parameters. The second parameter, *filename*, is required. The first parameter, *–options*, is optional. Options are preceded by a hyphen.

- In assembler syntax statements, column one is usually reserved for the first character of an **optional** label or symbol. If a label or symbol is a **required** parameter, the symbol or label will be shown starting against the left margin of the shaded box as in the example below. No instruction, command, directive, or parameter, other than a symbol or label, should begin in column one.

  symbol **.usect "***section name***",** *size in bytes*

The *symbol* is required for the .usect directive, and must begin in column one. The *section name* must be enclosed in quotes and the section *size in bytes* must be separated from the *section name* by a comma.

- Some directives can have a varying number of parameters. For example, the .byte directive can have up to 100 parameters. The syntax for this directive is:

Note that **.byte** does not begin in column one.

.byte $value_1$ [, ... , $value_n$]

This syntax shows that .byte must have at least one value parameter, but you have the option of supplying additional value parameters, separated by commas.

Information About Cautions

This is an example of a caution statement.

A caution statement describes a situation that could potentially damage your software or equipment.

Please read each caution statement carefully.

Related Documentation From Texas Instruments

The following books describe the TMS320C8x MVP and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

TMS320C80 Multimedia Video Processor Data Sheet (literature number SPRS023) describes the features of the ’C80 device and provides pinouts, electrical specifications, and timings for the device.

TMS320C80 Multimedia Video Processor (MVP) Technical Brief (literature number SPRU106) provides an overview of the ’C80 features, development environment, architecture, and memory organization.

TMS320C80 (MVP) C Source Debugger User's Guide (literature number SPRU107) describes the 'C80 master processor and parallel processor C source debuggers. This manual provides information about the features and operation of the debuggers and the parallel debug manager; it also includes basic information about C expressions and a description of progress and error messages.

TMS320C80 (MVP) Master Processor User's Guide (literature number SPRU109) describes the 'C80 master processor (MP). This manual provides information about the MP features, architecture, operation, and assembly language instruction set; it also includes sample applications that illustrate various MP operations.

TMS320C80 (MVP) Multitasking Executive User's Guide (literature number SPRU112) describes the 'C80 multitasking executive software. This manual provides information about the multitasking executive software features, operation, and interprocessor communications; it also includes a list of task error codes.

TMS320C80 (MVP) Parallel Processor User's Guide (literature number SPRU110) describes the 'C80 parallel processor (PP). This manual provides information about the PP features, architecture, operation, and assembly language instruction set; it also includes software applications and optimizations.

TMS320C80 (MVP) System-Level Synopsis (literature number SPRU113) contains the 'C80 system-level synopsis, which describes the 'C80 features, development environment, architecture, memory organization, and communication network (the crossbar).

TMS320C80 (MVP) Transfer Controller User's Guide (literature number SPRU105) describes the 'C80 transfer controller (TC).This manual provides information about the TC features, functional blocks, and operation; it also includes examples of block write operations for big- and little-endian modes.

TMS320C80 (MVP) Video Controller User's Guide (literature number SPRU111) describes the 'C80 video controller (VC). This manual provides information about the VC features, architecture, and operation; it also includes procedures and examples for programming the serial register transfer (SRT) controller and the frame timer registers.

If You Need Assistance. . .

If you want to. . .	Do this. . .
Request more information about Texas Instruments Digital Signal Processing (DSP) products	Write to: Texas Instruments Incorporated Market Communications Manager, MS 736 P.O. Box 1443 Houston, Texas 77251–1443
Order Texas Instruments documentation	Call the TI Literature Response Center: **(800) 477–8924**
Ask questions about product operation or report suspected problems	Call the DSP hotline: **(713) 274–2320** **FAX: (713) 274–2324**
Report mistakes in this document or any other TI documentation	Fill out and return the reader response card at the end of this book, or send your comments to: Texas Instruments Incorporated Technical Publications Manager, MS 702 P.O. Box 1443 Houston, Texas 77251–1443 Electronic mail: **comments@books.sc.ti.com**

Trademarks

SPARC is a trademark of SPARC International, Inc.

SunView, SunWindows, and Sun Workstation are trademarks of Sun Microsystems, Inc.

UNIX is a registered trademark of UNIX Systems Laboratories, Inc.

Contents

Figures

Tables

Examples

Chapter 1

C Compiler Description

The TMS320C8x (MVP) code generation tools package contains special shell utilities, mpcl and ppcl, that allow you to compile, assemble, and link your source files to create an executable object file with one command. This chapter provides a complete description of how to use these shell programs to compile, assemble, and link your programs.

The C compilers include an optimizer that allows you to produce highly optimized code. The optimizer is explained in Section 1.3, *Using the C Compiler Optimizer (–o Option).*

The C compilers translate ANSI standard C into assembly language. The ANSI standard identifies certain features that may differ from compiler to compiler. These differences are covered in Chapter 2, *The TMS320C8x C Language*.

The C compilers maintain a special runtime environment. If you write assembly language that interfaces with C, you need to maintain these environments as well. Chapters 3 and 4 explain the PP and MP runtime environments.

Topics

1.1 Compiling C Code

The mpcl and ppcl shell programs let you compile, assemble, and optionally link in one step. These shells run one or more source modules through the following:

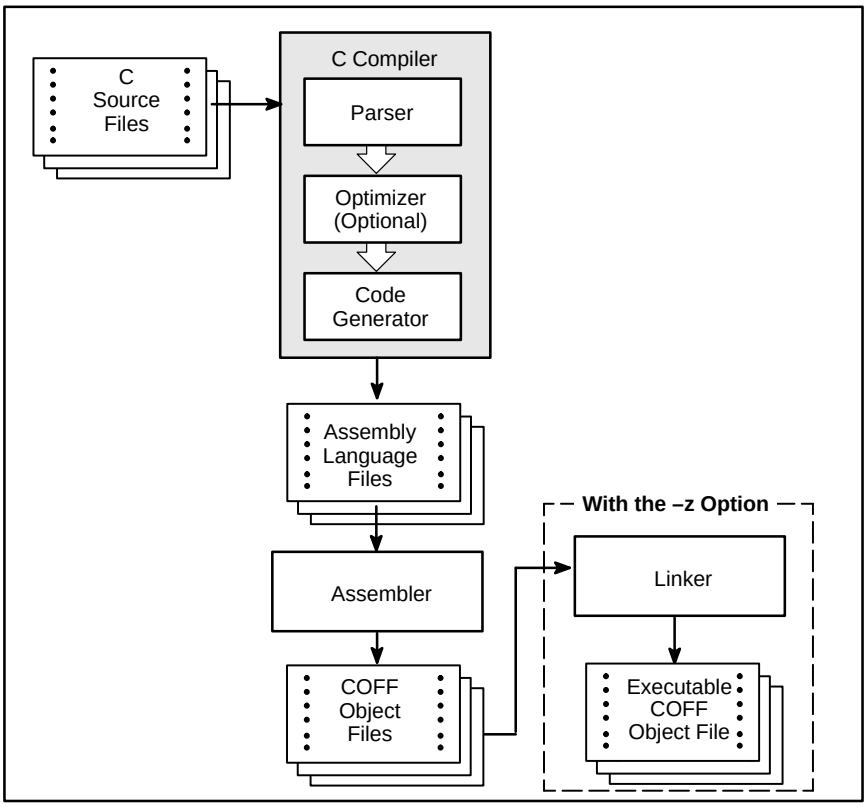
- The **compiler**, which includes the parser, optimizer, and code generator
- The **assembler**, which generates a COFF object file
- The **linker** (optional), which links your files to create an executable object file. The linker is optional at this point. You can compile and assemble various files with the shell and link later.

The C compiler shells can generate big-endian or little-endian code. Big-endian code is generated by default. See subsection 1.1.3, *Compiler Options,* for information on options to generate little-endian code.

For more information about the assembler, see Chapter 7, *Assembler Description,* and for more information about the linker, see Chapter 13, *Linker Description*.

By default, the shell compiles and assembles files; however, if you use the –z option, the shell also links your files. Figure 1–1 illustrates the path the mpcl or ppcl shell takes with and without the –z option.

Figure 1–1. The mpcl and ppcl Shell Program Overview



1.1.1 Invoking the C Compilers

To run the compilers, enter:

mpcl [*–options*] [*filenames*] [*–z* [*link_options*] [*object files*]]
or
ppcl [*–options*] [*filenames*] [*–z* [*link_options*] [*object files*]]

mpcl	is the command that invokes the TMS320C8x MP compiler and assembler.
ppcl	is the command that invokes the TMS320C8x PP compiler and assembler.
options	affect the way the compiler processes input files.
filenames	are one or more C source files, assembly source files, or object files.
–z	is the option that runs the linker.
link_options	control the linking process.
object files	names additional object files to link.

The options control the way the compiler processes files. The filenames provide a method of identifying source files, intermediate files, and output files. Options and filenames can be specified in any order on the command line. However, if you use the –z option, it and its associated information must follow all source filenames and compiler options.

The normal progress information consists of a banner for each compiler pass and the names of functions as they are defined. The example below shows the output from compiling a single module:

```
$ ppcl pptest
[pptest]
MVP PP ANSI C Compiler     Version x.xx
Copyright (c) 1993–1995    Texas Instruments Incorporated
   ”pptest.c”   ==> shellsort
MVP PP ANSI C Codegen      Version x.xx
Copyright (c) 1993–1995    Texas Instruments Incorporated
   ”pptest.c”:  ==> shellsort
MVP PP Macro Assembler     Version x.xx
Copyright (c) 1993–1995    Texas Instruments Incorporated
 PASS 1
 PASS 2

 No Errors, No Warnings
```

1.1.2 Specifying Filenames

The input files specified on the command line can be C source files, assembly source files, or object files. The mpcl and ppcl shells use filename extensions to determine the file type.

Extension	File Type
.asm or **.s*** (extension begins with s)	assembly language source file
.c or **no extension**	C source file
.o* (extension begins with o)	object file

Files without extensions are assumed to be C source files with a .c extension. For example, if you invoke the compiler with the command

```
ppcl xyz
```

the compiler will attempt to open a C source file called xyz.c.

You can use the –e option to change these default extensions, causing the shell to associate different extensions with assembly source files or object files. You can also use the –f option on the command line to override these file type interpretations for individual files. For more information about the –e and –f options, see *File specifiers*, page CG:1-11.

You can use wildcard filename specifications to compile multiple files. Wildcard specifications vary by system; use the appropriate form. To compile all the files in a directory, enter the following:

```
mpcl *.c  or
ppcl *.c
```

1.1.3 Compiler Options

Command line options control the operation of both the shell and the programs it runs. This section provides a general description of the conventions, a summary table, and a detailed description of each option.

Options consist of either single letters or two-letter pairs. They **are not** case sensitive, and they are preceded by a hyphen. Single-letter options with no parameters can be combined. For example, –sgq is equivalent to –s –g –q. Two-letter pairs that have the same first letter can be combined: for example, –me and –ma become –mea. Options that have parameters, such as –u *name* and –i *dir*, cannot be combined with other options.

You can set up default options for the shell by using the C_OPTION environment variable. For a detailed description of C_OPTION, see subsection 1.1.4, *Using the C_OPTION Environment Variable*.

Table 1–1 summarizes all shell and linker options. The table is followed by in-depth descriptions of each of the options.

Table 1–1. Options Summary Table

General Options	Option	Effect
These options control the overall operation of the ppcl or mpcl shell. For more information, see *General options,* page CG:1-9.	–c	disable linking (negates –z)
	–d*name*[=def]	predefine *name*
	–g	enable symbolic debugging
	–i*dir*	define #include search path
	–k	keep .asm file
	–n	compile only (do not assemble)
	–q	suppress progress messages (quiet)
	–qq	suppress all messages (super quiet)
	–s	interlist C and asm source statements
	–ss	interlist original **and** optimized C source with asm source statements
	–u*name*	undefine *name*
	–v2	set silicon version to PG2 support (default)
	–v3	set silicon version to PG3 support
	–z	enable linking (options described in linker options section of this table)
File Specifiers	**Option**	**Effect**
These options use the extensions of each filename to determine how to process the file. For more information, see *File specifiers*, page CG:1-11.	–ea	set default extension for assembly files
	–eo	set default extension for object files
	–fa*file*	identify assembly language file (default is .asm or .s*)
	–fc*file*	identify C source file (default is .c or no extension)
	–fo*file*	identify object file (default for .o*)
	–fr *pathname*	write object file to *pathname* directory
	–fs *pathname*	specify assembly file directory
	–ft *pathname*	override TMP environment variable
Parser Options	**Option**	**Effect**
These options control the preprocessing, syntax-checking, and error-handling behavior of the compiler. For more information, see *Parser options*, page CG:1-12.	–pe	treat code-E errors as warnings
	–pf	generate prototypes for functions
	–pk	allow K&R compatibility
	–pl	generate preprocessed listing (.pp) file
	–pn	suppress #line directives in .pp file
	–po	preprocess only
	–pr	generate error listing
	–pw	suppress warning messages
	–p?	enable trigraph expansion

Table 1–1. Options Summary Table (Continued)

Optimizer Options	Options	Effect
These options enable and control the optimization pass. For more information, see *Optimizer options*, page CG:1-14.	–o0	level 0 register optimization
	–o1	level 1, all level 0, plus local optimization
	–o2 (or –o)	level 2, all level 1, plus global optimization
	–o3	level 3, all level 2, plus file-level optimization
	–o*i**size*	set automatic inlining *size* (–o3 only)
	–ol0 (–oL0)	file alters a standard library function
	–ol1 (–oL1)	file defines a standard library function
	–ol2 (–oL2)	file does not define or alter library functions (default)
	–on0	disable the optimizer information file
	–on1	produce an optimizer information file
	–on2	produce a verbose information file
	–op0	functions in other files may call functions and modify variables defined here
	–op1	functions in other files do not call functions defined here, but may modify variables defined here (default)
	–op2	functions in other files do not call functions or modify variables defined here
Inlining Options	**Options**	**Effect**
These options control the inlining of functions. For more information, see *Inlining options*, page CG:1-16.	–x0	disable default inlining
	–x1	restore default inlining
	–x2 or –x	define _INLINE keyword
Type-Checking Options	**Options**	**Effect**
These options allow relaxed type-checking rules. For more information, see *Type-checking options*, page CG:1-16.	–tf	relax prototype checking
	–tp	relax pointer combination checking
Runtime Model Options	**Options**	**Effect**
These options are used to customize the executable output of the compiler for your specific application. For more information, see *Runtime-model options*, page CG:1-17.	–ma	assume aliased variables
	–mc	(PP only) leave constant data in external memory
	–me	generate little endian code
	–mz	don't unroll loops
Assembler Options	**Options**	**Effect**
These options control the assembler's behavior. For more information, see *Assembler options*, page CG:1-17.	–al	produce the assembly listing file
	–as	keep labels as symbols
	–ax	generate a cross-reference file

Table 1–1. Options Summary Table (Concluded)

Linker Options	Options	Effect
These options control the linker's behavior. For more information, see Chapter 13, *Linker Description*.	–a	generate absolute output
	–ar	generate relocatable output
	–c	use ROM initialization (MP)
	–cr	use RAM initialization (MP)
	–e*sym*	define entry point
	–f*val*	define fill value
	–g	keep C symbols global (overrides –h)
	–h	make global symbols static
	–heap*size*	set heap size (MP)
	–i*dir*	define library search path
	–l*lib*	supply library name
	–m*file*	name the map file
	–o*file*	name the output file
	–pc	use ROM initialization (PP)
	–pheap*size*	set heap size (PP)
	–pstack*size*	set stack size (PP)
	–r	generate relocatable output
	–s	strip symbol table
	–stack*size*	set stack size (MP)
	–t *sym*	perform task-level link
	–u*sym*	undefine *sym*
	–w	warn when creating a nonspecified output section
	–x	exhaustively read libraries

General options

You can use the general options described below to control the overall operation of the shell.

–c suppresses the linking option; it causes the shell not to run the linker even if –z is specified. This option is especially useful when you have –z specified in the C_OPTION environment variable and you don't want to link. For more information, see *Linker options*, page CG:1-17.

–d **–d***name*[**=***def*] predefines *name* for the preprocessor. This is equivalent to inserting **#define** *name def* at the top of each C source file that the shell encounters. If the optional [*def*] is omitted, *name* is defined as 1.

–g causes the compiler to generate symbolic directives for use with the TMS320C8x PP and MP C source level debuggers. Note that the use of –g will prevent the code generator from scheduling code across a C statement boundary. This can be very apparent in PP compiled code, due to fewer opportunities for generating parallel instructions and filling delay slots.

–i **–i***dir* adds *dir* to the list of directories the compiler searches for #include files. You can use this option a maximum of 64 times to define several directories; be sure to separate –i options with spaces. Note that if you don't specify a directory name, the preprocessor ignores the –i option. For more information, see subsection 1.2.2, *#include File Search Paths*.

–k keeps the assembly language file. Normally, the shell deletes the output assembly language file after compilation is finished, but using –k allows you to retain the assembly language output from the compiler.

–n causes the shell to compile only. If you use –n, the specified source files are compiled but not assembled or linked. This option overrides –z and –c options.

–q suppresses banners and progress information from **all** tools. Only source filenames and error messages are output.

–qq suppresses **all** output except error messages.

–s invokes the interlist utility, which interlists C source statements into the assembly language output of the compiler, allowing you to inspect the code generated for each C statement. This option implies the –k option. If you use the –s option with the optimizer, the interlist utility will interlist the assembly source with the *optimized* C code. To see both the original C and the optimized version, use –ss. For more information about the interlist utility see Section 1.5, *Using the Interlist Utility (–s and –ss Options)*.

–ss invokes the interlist utility. This option, when used in tandem with the optimizer, will interlist both the original C source, and the optimized C source, with the resulting assembly language source. For more information about the interlist utility see Section 1.5, *Using the Interlist Utility (–s and –ss Options)*.

–u **–u***name* undefines the predefined constant *name*. Overrides any –d options for *name*.

–v*n* set silicon version.

- **–v2** sets support for pre-production silicon version 2(default).
- **–v3** sets support for production silicon version 3.

–z enables the linking option; it causes the shell to run the linker on specified object files. The –z option must follow all source files and compiler options on the command line. All arguments that follow –z on the command line are passed to, and interpreted by, the linker. For more information, see subsection 1.7.2, *Using the Shell to Invoke the Linker (–z Option)*.

File specifiers

–e –ea [.]*asmext* for assembly files (default is .asm on the MP and .s on the PP)

–eo [.]*objext* for object files (default is .obj on the MP and .o on the PP)

For example:

```
ppcl -ea .rrr -eo .o37 fit.rrr
```

assembles the file fit.rrr and creates an object file named fit.o37.

The "." in the extensions and the space between the option and the extension are optional (the example could have been –earrr –eoo37…).

The –e option affects both the interpretation of filenames on the command line and the names of the output files and should always precede any filename on the command line.

–f overrides default interpretations for source file extensions. If your naming conventions do not conform to those of the shell, you can use –f options to specify which files are C source files, assembly language files, and object files. You can insert an optional space between the –f option and the filename. The –f options are:

–fa*file* for assembly language source files

–fc*file* for source files

–fo*file* for object files

If you have a C source file called cfile.s and an assembly language file called assy, use –f to force the correct interpretation:

```
mpcl –fc cfile.s –fa assy
```

Note that –f cannot be applied to a wildcard file specification.

–fr permits you to specify a directory for object files. If the –fr option is not specified, the shell will place object files in the current directory. To specify an object file directory, insert the directory's pathname on the command line after the –fr option:

```
mpcl –fr /home/object
```

–fs permits you to specify a directory for assembly files generated by the compiler. If the –fs option is not specified, the shell will place assembly files in the current directory. To specify an assembly file directory, insert the directory's pathname on the command line after the –fs option:

```
mpcl –fs /home/assembly
```

–ft permits you to specify a directory for temporary intermediate files generated by the compiler. The –ft option overrides the TMP environment variable (described in subsection 1.1.5, *Using the TMP Environment Variable*). To specify a temporary directory, insert the directory's pathname on the command line after the –ft option:

```
mpcl –ft /tmp ...
```

Parser options

–pe causes the parser to treat code-E errors as warnings, allowing complete compilation. Normally, the code generator will not run if the parser detects any code-E errors. Note that the code-F errors are always fatal. For more information, see Section 1.6, *How the Compiler Handles Errors*.

–pf helps you create files containing ANSI-style function prototypes for your program. The –pf option causes the parser to write out a *.pro* file that contains a prototype declaration for every function definition in the C file.

–pk relaxes certain requirements imposed by the ANSI C standard (that are stricter than those required by earlier Kernighan and Ritchie compilers). This facilitates compatibility between Kernighan and Ritchie programs and the ANSI compiler. The effects of the –pk option are further described in Section 2.11, *Compatibility With K&R C*.

–pl generates a preprocessed listing file. The compiler writes a modified version of the source file to an output file called file.pp. The .pp file contains all the source from #include files and the expanded macros; it does not contain any comments or code for false #if or #ifdef directives. The only remaining preprocessor directive is #line. For more information, see subsection 1.2.3, *Generating a Preprocessed Listing File (–pl Option)*.

–pn suppresses line and file information. –pn causes the #line directives of the form

#123 file.c.

to be suppressed in the .pp file generated when the –po or –pl options are used. –pn is sometimes useful when you are compiling machine-generated source.

–po runs the compiler for preprocessing only. When invoked with the –po option, the compiler processes only macro expansions, #include files, and conditional compilation. The compiler writes the preprocessed file out with a .pp extension. For more information, see subsection 1.2.3, *Generating a Preprocessed Listing File (–pl Option)*.

–pr creates a parser error message file. The error file will have the base name of the input file and an .err extension. The file will contain all error messages generated by the parser.

–pw suppresses warning messages (code-W errors). Rather than producing warning messages, the compiler produces diagnostic messages for only those errors that prevent complete compilation. This option can be repeated (–pww) to suppress recoverable messages (code-E errors). For more information, see Section 1.6, *How the Compiler Handles Errors*.

–p? enables trigraph expansion. Trigraphs are special escape sequences of the form

```
??c
```

(where c is a character). The ANSI C standard defines these sequences for the purpose of compiling programs on systems with limited character sets. By default, the compiler does not recognize trigraphs; use –p? to enable trigraphs. For more information, refer to the ANSI C specification, § 2.2.1.1.

Optimizer options

–o*n* causes the compiler to optimize the intermediate file that is produced by the parser. *n* denotes the level of optimization. There are four levels of optimizations: **–o0**, **–o1**, **–02**, and **–o3**.

If you do not indicate a level (0, 1, 2, 3) after the –o option, the optimizer defaults to level 2 optimization. For more information about the optimizer, see Section 1.3, *Using the C Compiler Optimizer (–o Option)*.

–oi*size* controls automatic inlining of functions at optimization level 3. You specify the *size* limit for the largest function that will be inlined. If the –oi option is not used, the optimizer will inline very small functions when invoked at level 3. Setting the size to 0 (–oi0) disables automatic inlining completely. The –x option controls inlining of functions declared with the keyword inline.

–ol*n* (lowercase L) controls file level optimizations. When you invoke the optimizer at level 3 (–o3), some of the optimizations make use of known properties of the standard library functions. If the file you are compiling redefines any of these standard functions, the compiler may produce incorrect code. Use the –ol option to notify the optimizer if any of the following situations exist:

- **–ol0**: This file defines a function with the same name as a standard library function.
- **–ol1**: This file contains the standard library definition functions for those functions that are defined in it.
- **–ol2**: This file does not alter standard library functions. Use this option to restore the default behavior of the optimizer if you have used one of the other two options in a command file, an environment variable, etc.

–on*n* causes the compiler to produce a user readable optimization information file with a .nfo extension. This option works only when used with the –o3 option. There are three levels available:

- **–on0**: Do not produce an information file. Use this option to restore the default behavior of the optimizer if you have used one of the other two options in a command file, an environment variable, etc.
- **–on1**: Produce an optimization information file.
- **–on2**: Produce a verbose optimization information file.

–op*n* controls some of the optimizer's level 3 optimizations. At level 3 (–o3), the optimizer can make optimizations more effectively if it knows how to treat variables declared as external (EXTERN). There are three levels available:

- **–op0**: Signals the optimizer that functions in other files might call functions or variables defined in the current file. This disables some of the –o3 optimizations.
- **–op1**: Signals the optimizer that no functions exist in other files that might call functions defined in this file and that no interrupt function defined elsewhere might call functions defined here. This is the default when –o3 is used.
- **–op2** or **–op**: Signals the optimizer that no functions in this module are called by other modules and no variable declared in this module will be altered by another module.

Level –op1 reverts to –op0 if the file does not define the function *main* or an interrupt function or if it contains calls to unknown functions. This level is the default because it is unlikely that an interrupt function defined elsewhere would call a user function defined in a file that contains main and all of the user functions called directly or indirectly from main.

Level –op2 also reverts to –op0 if no *main* or interrupt functions are found, but unlike level 1, it does not revert if there are calls to unknown functions.

Use of the –op options automatically invokes the optimizer at level 3 (–o3).

Inlining options

–x*n* controls definition-controlled function inlining done by the optimizer. The options are:

–x0 disables default inlining

–x1 inlines all intrinsic operators

–x2/–x invokes the optimizer at level 2 and defines the _INLINE preprocessor symbol. This option may be specified as –x or –x2 interchangeably.

See Section 1.4, *Inline Function Expansion* for more details.

Type-checking options

–tf relaxes type-checking on redeclaration of prototyped functions. In ANSI C, if a function is declared with an old-format declaration, such as

```
int func();
```

and then later declared with a prototype, such as

```
int func(float a, char b);
```

this generates an error because the parameter types in the prototype disagree with the default argument promotions (which convert float to double and char to int). With the –tf option, the compiler overlooks such redeclarations of parameter lists.

–tp relaxes type-checking on pointer combinations. This option has two effects:

A pointer to a signed type can be combined in an operation with a pointer to the corresponding unsigned type:

```
int *pi; unsigned *pu;
pi = pu;        /*Illegal unless -tp used */
```

Pointers to differently qualified types can be combined:

```
char *p; const char *pc;
p = pc;         /*Illegal unless -tp used */
```

–tp is especially useful when pointers are passed to prototyped functions and when the passed pointer type would ordinarily disagree with the declared parameter type in the prototype.

Runtime-model options

–ma assumes variables are aliased. The compiler assumes that pointers may alias (point to) named variables and therefore aborts register optimizations when an assignment is made through a pointer.

–mc (PP only) places string constants and switch tables in external memory. Without this option, string constants and switch tables are placed in on-chip locations. Placing this data in external memory frees on-chip memory for other uses, but doing so results in slower access.

–me generates code for little-endian configurations. By default, the compiler generates big-endian code.

–mz prevents the optimizer from unrolling loops. Useful if unrolling causes loops to become too large for a cache block or if it causes register spilling.

For more information about optimization, see Section 1.3, *Using the C Compiler Optimizer (–o Option)*.

Assembler options

–al invokes the assembler with the –l (lowercase L) option to produce an assembly language listing file.

–as retains labels. Label definitions are written to the COFF symbol table for use with symbolic debugging.

–ax invokes the assembler with the –x option to produce a symbolic cross-reference in the listing file.

For more information about assembler options, see Chapter 7, *Assembler Description*.

Linker options

Linker options can be used with the compiler shell or with the linker as a standalone. (See Section 1.7, *Linking C Code*.) When used with the compiler shell, all linker options should follow the –z option described on page CG:1-10. For example:

```
mpcl   –q symtab.c –z –c –o symtab.out –l mp_rts.lib
```

In this example, the file symtab.c will be compiled with the –q (quiet) option. The –z option causes the shell to invoke the linker and pass the –c, –o, and –l linker options to the linker.

All shell command line options following –z are passed to the linker. For this reason, the –z option followed by the linker options must be the last shell options specified on the command line.

The –c and –n compiler options suppress the –z linker option and cause the shell not to run the linker even if –z has been specified (see *General options*, page CG:1-9 for more information). The Linker Options section of Table 1–1, *Options Summary Table*, beginning on page CG:1-6, summarizes the linker options. For more information about linker options, see Chapter 13, *Linker Description*.

1.1.4 Using the C_OPTION Environment Variable

An environment variable is a system symbol that you define and assign a string to. You may find it useful to set the shell default options using the C_OPTION environment variable; if you do, these default options and/or input filenames are used every time you run the shell.

Setting up default options with the C_OPTION environment variable is especially useful when you want to run the shell consecutive times with the same set of options and/or input files. After the shell reads the command line and the input filenames, it reads the C_OPTION environment variable and processes it.

Options specified with the environment variable are specified in the same way and have the same meaning as they do on the command line.

For example, if you want to always run quietly, enable symbolic debugging, and link, then set up the C_OPTION environment variable as follows.

```
setenv C_OPTION "–qg –z"
```

You may want to set C_OPTION in your system initialization file; for example, in your .login or .cshrc file.

Using the –z option enables linking. If you plan to link most of the time when using the shell, you can specify the –z option with C_OPTION. Later, if you need to invoke the shell without linking, you can use –c on the command line to override the –z specified with C_OPTION. These examples assume C_OPTION is set as shown previously:

```
ppcl *.c             ; compiles and links
ppcl –c *.c          ; only compiles
ppcl *.c –z c.cmd    ; compiles/links using a command file
ppcl –c *.c –z c.cmd ; only compiles (–c overrides –z)
```

1.1.5 Using the TMP Environment Variable

The mpcl or ppcl shell program creates intermediate files as it processes your program. For example, the parser phase of the shell creates a temporary file used as input by the code generation phase. By default, the shell puts intermediate files in the current directory. However, you can name a specific directory for temporary files.

This feature allows use of a RAM disk or other high-speed storage files. It also allows source files to be compiled from a remote directory without writing any files into the directory where the source resides. This is useful for protected directories.

There are two ways to specify a temporary directory:

- Use the TMP environment variable:

  ```
  setenv TMP ”/tmp”
  ```

- Use the –ft option on the command line:

  ```
  ppcl –ft /tmp
  ```

  The –ft option, if used, overrides the TMP environment variable.

1.2 Controlling the Preprocessor

The TMS320C8x C compilers include standard C preprocessing functions, which are built into the first pass of the compiler. The preprocessor handles:

- Macro definitions and expansions
- #include files
- Conditional compilation
- Various other preprocessor directives (specified in the source file as lines beginning with the # character)

This section describes specific features of the TMS320C8x preprocessor. A general description of C preprocessing is in Section A12 of *The C Programming Language* by Kernighan and Ritchie.

1.2.1 Predefined Names

The TMS320C8x C compilers maintain and recognize the predefined macro names listed below:

__DATE__	expands to the compilation date, in the form *“mm dd yyyy”*.
__FILE__	expands to the current source filename.
__LINE__	expands to the current line number.
__STDC__	expands to 1 (identifies the compiler as ANSI standard).
__TIME__	expands to the compilation time, in the form *“hh:mm:ss”*.
_INLINE	expands to 1 if the –x or –x2 optimizer option is used. It is undefined otherwise.
_MVP_MP	expands to 1 if mpcl is used. It is undefined otherwise.
_MVP_MP_BIG	expands to 1 if mpcl is used without the –me option. It is undefined otherwise.
_MVP_MP_LITTLE	expands to 1 if mpcl is used with the –me option. It is undefined otherwise.
_MVP_PP	expands to 1 if ppcl is used. It is undefined otherwise.
_MVP_PP_BIG	expands to 1 if ppcl is used without the –me option. It is undefined otherwise.
_MVP_PP_LITTLE	expands to 1 if ppcl is used with the –me option. It is undefined otherwise.

You can predefine additional names from the command line by using the –d option:

```
ppcl –dNAME –dREGS=6 *.c
```

This has the same effect as including these lines at the beginning of each source file:

```
#define NAME 1
#define REGS 6
```

1.2.2 #include File Search Paths

The #include preprocessor directive tells the compiler to read source statements from another file. The syntax for this directive is:

#include "*filename***"** or **#include <***filename***>**

The *filename* names an #include file that the compiler reads statements from; you can enclose the filename in double quotes or in angle brackets. The filename can be a complete pathname, have partial path information, or have no path information.

- If you enclose the filename in *double quotes*, the compiler searches directories for the file in the following order:
  1) The directory that contains the current source file. (The current source file refers to the file that is being compiled when the compiler encounters the #include directive.)
  2) Any directories named with the –i compiler option in the mpcl or ppcl shell
  3) Any directories set with the environment variable C_DIR
- If you enclose the filename in *angle brackets*, the compiler searches directories for the file in the following order:
  1) Any directories named with the –i option in the mpcl or ppcl shell
  2) Any directories set with the environment variable C_DIR

Note: Compiler Search Patterns for Input Files

If you enclose the filename in angle brackets (<>), the compiler **does not search** for the file in the current directory.

The –i shell option

The –i shell option names an alternate directory that contains #include files. The format of the –i option is:

ppcl –i *pathname* ... or
mpcl –i *pathname* ...

You can use up to 64 –i options per invocation; each –i option names one *pathname*. In C source, you can use the #include directive without specifying path information for the file; instead, you can specify the path information with the –i option. For example, assume that a file called source.c is in the current directory. The source.c file contains one of the following directive statements:

```
#include ”alt.h” or
#include <alt.h>
```

If the path for alt.h is

```
/mvp/files/alt.h
```

use the command:

```
mpcl –i/mvp/files source.c
```

to invoke the mp shell.

C_DIR environment variable

The compiler uses the environment variable **C_DIR** to name alternate directories that contain #include files. To specify the same directory for #include files as in the previous example, set C_DIR with the command:

```
setenv C_DIR ”/mvp/files”
```

Then you can include alt.h:

```
#include ”alt.h” or
#include <alt.h>
```

and invoke the compiler without the –i option:

```
ppcl source.c
```

This results in the compiler using the path in the environment variable to find the #include file.

The pathnames specified with C_DIR are directories that contain #include files. You can separate pathnames with a semicolon or with blanks. In C source, you can use the #include directive without specifying any path information; instead, you can specify the path information with C_DIR.

The environment variable remains set until you reboot the system or reset the variable by entering:

```
unsetenv C_DIR
```

1.2.3 Generating a Preprocessed Listing File (–pl Option)

The –pl shell option allows you to generate a preprocessed version of your source file. The compiler's preprocessing functions perform the following on the source file:

- Each source line ending in backslash (\) is joined with the line that follows.
- Trigraph sequences are expanded (if enabled with the –p? option).
- Comments are removed.
- #include files are copied into the file.
- Macro definitions are processed, and all macros are expanded.
- All other preprocessing directives, including #line directives and conditional compilation, are executed.

(These functions correspond to translation phases 1–3 as specified in section A12 of K&R.)

The preprocessed output file contains no preprocessor directives other than #line; the compiler inserts #line directives to synchronize line and file information in the output files with input position from the original source files. If you use the –pn option, no #line directives are inserted.

If you use the –po option, the compiler performs *only* the preprocessing functions listed above and then writes out the preprocessed listing file; no syntax checking or code generation takes place. The –po option can be useful when you debug macro definitions or when host memory limitations dictate separate preprocessing. The resulting preprocessed listing file is a valid C source file that can be rerun through the compiler.

1.2.4 #error and #warn Directives

The standard #error preprocessor directive forces the compiler to issue a diagnostic message and halt compilation. The compiler extends the #error directive with a #warn directive, which, like #error, forces a diagnostic message but does not halt compilation. The syntax of #warn is identical to that of #error. For more information, refer to Section A12.7 of K&R.

1.3 Using the C Compiler Optimizer (–o Option)

The compiler package includes an optimization program that improves the execution speed and reduces the size of C programs by simplifying loops, rearranging statements and expressions, and allocating variables into registers.

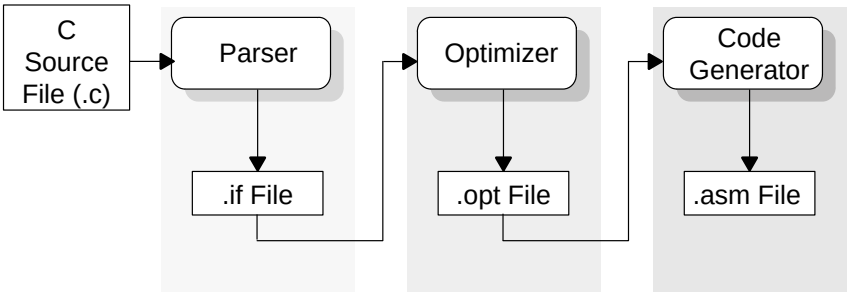
The optimizer runs as a separate pass between the parser and the code generator. The easiest way to invoke the optimizer is to use the mpcl or ppcl shell program, specifying the –o option on the command line. The –o option may be followed by a digit specifying the level of optimization. The default level is 2; the highest level is 3.

For example, to invoke the compiler using level 2 optimization, enter:

```
mpcl –o function.c  or
ppcl –o function.c
```

Figure 1–2 illustrates the execution flow of the compiler with optimization enabled.

Figure 1–2. Compiling a C Program with the Optimizer



1.3.1 Optimization Levels

There are four levels of optimization: 0, 1, 2, and 3. These levels control the type and degree of optimization, as described in the following list:

- **Level 0**
  - performs control-flow-graph simplification
  - allocates variables to registers
  - performs loop rotation
  - eliminates dead code
  - simplifies expressions and statements
  - expands calls to functions declared inline

❑ **Level 1**

performs all level 0 features, plus:

- performs local copy/constant propagation
- removes dead assignments
- eliminates local common expressions

❑ **Level 2**

performs all level 1 features, plus:

- performs loop optimizations
- eliminates global common subexpressions
- eliminates global redundant assignments
- converts array references in loops to incremented pointer form
- performs loop unrolling

❑ **Level 3**

performs all level 2 features, plus;

- removes all functions that are never called
- simplifies functions with return values that are never used
- expands calls to small functions inline
- reorders function definitions so that the attributes of called functions are known when the caller is optimized
- propagates arguments into function bodies when all call sites pass the same value in the same argument position
- identifies file-level variable characteristics

Note: Files That Redefine Standard Library Functions

The optimizer uses known properties of the standard library functions to perform level 3 optimizations. If you have files that redefine standard library functions, use the –ol (lowercase L) options to inform the optimizer. (See *Optimizer options,* page CG:1-14.)

The above list describes optimizations performed by the standalone optimization pass. The code generator performs several additional optimizations, particularly TMS320C8x-specific optimizations; it does so regardless of whether you invoke the optimizer. These optimizations are always enabled and are not affected by the optimization level you choose.

1.3.2 Debugging Optimized Code

When debugging a program, ideally you should debug it before using the optimizer and reverify its correctness after it has been optimized. The debugger may be used with optimized code, but the extensive rearrangement of code and the many-to-one allocation of variables to registers often makes it difficult, for both you and the debugger, to correlate source code with object code.

Note: Symbolic Debugging and Optimized Code

If you use the –g option to generate symbolic debugging information, many code generator optimizations are disabled because they disrupt the debugger.

1.3.3 Special Considerations When Using the Optimizer

The optimizer is designed to improve your ANSI-conforming C programs while maintaining their correctness. However, when you write code for the optimizer, you should note the following special considerations to insure that your program performs as you intend.

Using asm statements

You must be extremely careful when using asm (inline assembly) statements in optimized code. The optimizer rearranges code segments, uses registers freely, and may completely remove variables or expressions. Although the compiler will never optimize out an asm statement (except when it is totally unreachable), the surrounding environment where the assembly code is inserted may differ significantly from its apparent context in the C source code. It is usually safe to use asm statements to manipulate hardware controls such as interrupt registers or I/O ports, but asm statements that attempt to interface with the C environment or access C variables may have unexpected results. After compilation, check the assembly output to make sure your asm statements are correct and maintain the integrity of the program.

The volatile keyword

The optimizer analyzes data flow to avoid memory accesses whenever possible. If you have code that **depends** on memory accesses exactly as written in the C code, you **must** use the volatile keyword to identify these accesses. The compiler won't optimize out any references to volatile variables.

In the following example, the loop waits for a location to be read as 0xFF:

```
unsigned int *ctrl;
while (*ctrl !=0xFF);
```

In this example, *ctrl is a loop-invariant expression, so the loop will be optimized down to a single memory read. To correct this, declare ctrl as:

```
volatile unsigned int *ctrl;
```

Aliasing

Aliasing occurs when a single object may be accessed in more than one way, such as when two pointers point to the same object or when a pointer points to a named object. Aliasing can disrupt optimization because any indirect reference could potentially refer to any other object. The optimizer analyzes the code to determine where aliasing can and cannot occur, then optimizes as much as possible while still preserving the correctness of the program.

The compiler assumes that if the address of a local variable is passed to a function, the function might change the local variable by writing through the pointer, but that it will not make its address available for use elsewhere after returning. For example, the called function cannot assign the local's address to a global variable or return it. In cases where this assumption is invalid, use the –ma option in ppcl or mpcl to force the compiler to assume worst-case aliasing. In worst-case aliasing, any indirect reference may refer to such a variable.

Register use

When you compile with the optimizer, the compiler's register usage conventions allow more variables to be stored in registers. This should not affect the correctness of normal C code, but it may invalidate some asm statements that assume that certain variables are in certain registers. Also, interrupt service routines must save all registers that may be used by C functions. The register conventions and requirements for interrupt functions are described in Section 3.3, *PP Register Conventions* and Section 4.4, *MP Register Conventions*.

1.4 Inline Function Expansion

It is sometimes advantageous to replace a call to a short function with an inline expansion of the function's code. Function inlining can be advantageous in short functions for two reasons:

- It saves the overhead of a function call.
- Once inlined, the optimizer is free to optimize the function in context with the surrounding code.

The compiler will automatically expand the intrinsic operators of the target system (such as abs) by default. This happens whether or not the optimizer is used and whether or not any compiler or optimizer options are used. (You can defeat this automatic inlining by invoking the compiler with the –x0 option.) Functions that expand to intrinsic operators are:

- abs
- labs
- fabs

In addition, when the optimizer is invoked, the compilers perform two other types of inline function expansion: automatic and definition controlled.

1.4.1 Automatic inline Expansion (The –oi*size* Option)

Automatic inline expansion is done when the compiler is invoked with level 3 optimization. By default with this level of optimization, the compiler will inline very small functions automatically. You can change the size of functions that are automatically inlined with the –**oi***size* option. The **–oi***size* option specifies that functions whose size is less than *size* units are inlined regardless of how they were declared. The compiler measures the size of a function in arbitrary units; however, the size of each function is reported in the optimizer information file (–on1 option). If you want to be certain that a function is always inlined, use the inline keyword (discussed in the next subsection). You can defeat all automatic inlining by setting the size to 0 (–oi0).

- If you set the size parameter to 0 (–oi0), all size-controlled inlining is disabled.
- If you do not use the –oi, the optimizer inlines very small functions.

- If you set the size parameter to a nonzero integer, the compiler will inline all functions whose size is less than the size parameter. If the function is called more than once, the compiler multiplies the size of the function by the number of calls, and will inline the function only if the resulting product is less than the size parameter. The compiler measures the size of a function in arbitrary units. The optimizer information file (created with the –on1 or –on2 option), however, will report the size of each function in the same units that the –oi option uses.

1.4.2 Definition-Controlled Inline Function Expansion

Definition-controlled inline expansion is performed when the inline keyword is encountered in source code *and* the compiler is invoked with optimization. Functions with local static variables or a variable number of arguments will not be inlined. In addition, a limit is placed on the depth of inlining for recursive or non-leaf functions. Inlining should be used for small functions or functions that are called only once (though the compiler does not enforce this.) You can control this type of function inlining two ways:

- By *defining* a function as inline within a module (with the inline keyword), you can specify that the function is inlined *within that module*. A global symbol for the function is created, but the function will be inlined only within the module where it is defined as inline. Since the function is not visible within other modules, the function is called normally from other modules, rather than inlined.

- By *declaring* a function as static inline (typically in a header file), you can specify that the function is inlined in any module where it is visible (typically, in any module that includes the header). When you do this, you will probably want to use the –x option and the _INLINE keyword to control its declaration. If you fail to do this and subsequently compile *without* the optimizer, the call to the function will be unresolved.

The inline keyword

The keyword *inline* specifies that a function is expanded inline at the point at which it is called rather than by using standard calling procedures.

The inline keyword can be used in two ways:

inline *return-type function-name* (*parameter declarations*) { *function* }

creates a *definition* of the function so that a global symbol for the function is defined and code for the function is generated. It also specifies that all calls to the function in the current source module will be inlined.

static inline *return-type function-name* (*parameter declarations*) {*function*}

specifies that the function is to be expanded inline and that no code is generated for the function declaration itself. This usage is a *declaration* of the function, and **not** a definition. Functions declared in this way may be placed in header files and included by all source modules of the program. Use the –x option and the _INLINE preprocessor symbol to control its declaration.

Defining the _INLINE preprocessor symbol (–x option)

A command line switch controls the definition of the _INLINE keyword and some types of inline function expansion.

–x0 no inline expansion. Defeats the default expansions of the intrinsic operator functions.

–x1 resets the default behavior. You will use this option primarily to reset the default behavior from the command line if you have used another –x option in an environment variable or command file.

–x2 or **–x** creates the preprocessor symbol _INLINE, assigns it the value 1, and, if the optimizer was not invoked with a separate command line option, invokes the optimizer at the default level (–o2).

The _INLINE preprocessor symbol is defined (and set to 1) if the parser (or compiler shell utility) is invoked with the –x2 (or –x) option. It allows you to write code so that it will run whether or not the optimizer is used. It is used by standard header files included with the compiler to control the declaration of standard C runtime functions.

Example 1–1 on page CG:1-31 illustrates how the runtime-support library uses the _INLINE symbol.

The _INLINE symbol is used in the string.h header file to declare the function correctly, regardless of whether inlining is used. The _INLINE symbol is used to conditionally define _ _INLINE so that strlen is declared as static inline only if the _INLINE symbol is defined.

If the rest of the modules are compiled with inlining enabled *and* the string.h header is included, all references to strlen will be inlined and the linker will not have to use the strlen in the runtime-support library to resolve any references. Otherwise, the run-time-support library code will be used to resolve the references to strlen and function calls will be generated.

You will want to use the _INLINE preprocessor symbol in the same way that the function libraries use it, so that your programs will run regardless of whether inlining mode is selected for any or all of the modules in your program.

Example 1–1. How the Runtime Support Library Uses the _INLINE Symbol

```
/*****************************************************************************/
/* string.h   v1.00                                                          */
/*****************************************************************************/

#if _INLINE
#define __INLINE static inline
#else
#define __INLINE
#endif

__INLINE size_t  strlen(const char *_string);

#if _INLINE

/*****************************************************************************/
/*  strlen                                                                   */
/*****************************************************************************/
static inline size_t strlen(const char *string)
{
   size_t      n = –1;
   const char *s = string – 1;

   do n++; while (*++s);
   return n;
}

#endif
```

1.5 Using the Interlist Utility (–s and –ss Options)

The compiler package includes a utility that interlists C source statements into the assembly language output of the compiler. The interlist utility enables you to inspect the assembly code generated for each C statement. The interlist utility behaves differently depending on whether or not the optimizer is being used.

The easiest way to invoke the interlist utility is to use the –s shell option. To compile and run the interlist utility on a program called function.c, enter:

```
mpcl –s function
```
or

```
ppcl –s function
```

The interlist utility without the optimizer

The interlist utility runs as a separate pass between the code generator and the assembler. It reads both the assembly and C source files, merges them, and writes the C statements into the assembly file as comments beginning with ;>>>>. The output assembly file, function.asm, is assembled normally. The –s option automatically prevents the mpcl or ppcl shell from deleting the interlisted assembly language file (as if you had used –k).

If the interlist utility is used without the optimizer running, the –s option will have the same effect as the –g option on scheduling. That is, it will prevent the code generator from scheduling code across a C statement boundary. This will limit parallelism in the PP and also will prevent optimal delay slot filling in both the MP and the PP.

Example 1–2 shows a typical interlisted assembly file.

Example 1–2. An Interlisted MP Assembly Language File

```
sp  .set   r1
;  mpac test test.if
   .global   _binsearch
;>>>>  int binsearch(int x, int v[], int n)
;*************************************************************
;
;* FUNCTION DEF: _binsearch                                   *
;*************************************************************
;
_binsearch:
         addu      –24,sp,sp
;* r2    assigned to _x
;* r4    assigned to _v
;* r6    assigned to _n
;>>>>     int low, high, mid;
         st        8(sp),r6
         st        4(sp),r4
         st        0(sp),r2
;>>>>     low = 0;
         st        12(sp),r0
;>>>>     high = n–1;
         ld        8(sp),r2
         addu      –1,r2,r2
         st        16(sp),r2
;>>>>     while (low<=high) {
         ld        12(sp),r3
         ld        16(sp),r2
         cmp       r3,r2,r2
         bbo.a     L8,r2,gt.w
; branch occurs
L2:
;>>>>         mid = (low+high) / 2;
         ld        16(sp),r3
         ld        12(sp),r2
         addu      r3,r2,r2
         srl       31,32,r2,r3
         addu      r3,r2,r2
         sra       1,32,r2,r2
         st        20(sp),r2
;>>>>         if (x < v[mid])
         ld        4(sp),r4
         ld        20(sp),r3
         ld        0(sp),r2
         ld        r3:s(r4),r3
         cmp       r2,r3,r2
         bbo.a     L4,r2,ge.w
; branch occurs
;>>>>             high = mid –1;
         ld        20(sp),r2
         addu      –1,r2,r2
         st        16(sp),r2
         br.a      L7
; branch occurs
   .
   .
   .
```

The interlist utility with the optimizer

If the –s option is used with the optimizer (–o), the interlist utility does **not** run as a separate pass. Instead, the optimizer will insert comments into the code indicating how the optimizer has rearranged and optimized the code. These comments appear in the assembly language file as comments starting with ;***.

To view both the optimizers comments, and the original C source interlisted with the assembly code, use the –ss option. With the –ss option, the optimizer will insert its comments, and the interlist utility will run between the code generator and the assembler, merging the original C source into the assembly file.

Example 1–3 shows the function from Example 1–2 compiled with the optimizer (–o2) and –s.

Example 1–3. The Function From Example 1–2 Optimized

```
sp  .set   r1
;   mpac test test.if
;   mpopt –s –O2 –e test.if test.opt
    .global   _binsearch
;*************************************************************
;* FUNCTION DEF: _binsearch                                  *
;*************************************************************
_binsearch:
;*** 5 ----------------------    high = n–1;
;*** 6 ----------------------    if ( n <= 0 ) goto g9;
          move      r2,r3
          bcnd      L8,r6,le0.w
          addu      –1,r6,r5
; branch occurs
;*** 4 ----------------------    low = 0;
          move      r0,r6
L3:
;***   ----------------------g3:
;*** 7 ----------------------    mid = (low+high)/2;
;*** 8 ----------------------    C$1 = v[mid];
;*** 8 ----------------------    if ( x < C$1 ) goto g7;
          addu      r5,r6,r2
          srl       31,32,r2,r7
          addu      r7,r2,r2
          sra       1,32,r2,r2
          ld        r2:s(r4),r7
          cmp       r3,r7,r8
          bbo.a     L6,r8,lt.w
; branch occurs
;*** 10   ----------------------    if ( x > C$1 ) goto g6;
    .
    .
    .
```

1.6 How the Compiler Handles Errors

One of the compiler's primary functions is to detect and report errors in the source program. When the compiler encounters an error in your program, it displays a message in the following format:

"*file.c***", line** *n***: [***ECODE***]** *error message*

"file.c"	identifies the filename.
line *n*:	identifies the line where the error occurs.
[ECODE]	is a 4-character error code. A single upper-case letter identifies the error class; a 3-digit number uniquely identifies the error.
error message	is the text of the message.

Errors are divided into four classes according to severity; these classes are identified by the letters W, E, F, and I (upper-case i).

- **Code-W errors** are warnings. They result from a condition that is technically undefined according to the rules of the language, and code may not generate what you intended. This is an example of a code-W error:

```
"file.c", line 42: [W063] illegal type for register variable 'x'
```

- **Code-E errors** are recoverable. They result from a condition that violates the semantic rules of the language. Although these errors are normally fatal, the compiler can recover and generate an output file if you use the –pe option. For more information, see subsection 1.6.1, *Treating Code-E Errors as Warnings (–pe Option)*. This is an example of a code-E error:

```
"file.c", line 66: [E056] illegal storage class for function 'f'
```

- **Code-F errors** are always fatal. They result from a condition that violates the syntactic or semantic rules of the language. The compiler cannot recover and therefore does not generate output for code-F errors. This is an example of a code-F error:

```
"file.c", line 71: [F090] structure member 'a' undefined
```

- **Code-I errors** are implementation errors. They occur when one of the compiler's internal limits is exceeded. These errors are usually caused by extreme behavior in the source code rather than by explicit errors. In most cases, code-I errors cause the compiler to abort immediately. Most code-I messages contain the maximum value for the limit that was exceeded. (Those limits that are absolute are also listed in Table 2–4, *Absolute Compiler Limits*.) This is an example of a code-I error:

```
"file.c", line 99:[I015] block nesting too deep (max=20)
```

- **Other errors** are also reported, such as incorrect command line syntax or inability to find specified files. These errors are usually fatal and are identified by the symbol >> preceding the message. This is an example of such an error:

```
>> Cannot open source file 'mystery.c'
```

1.6.1 Treating Code-E Errors as Warnings (–pe Option)

A *fatal error* is an error that prevents the compiler from generating an output file. Normally, code-E, -F, and -I errors are fatal, while -W errors are not. The –pe shell option causes the compiler to effectively treat code-E errors as warnings so that the compiler will generate code for the file (despite the error).

Using –pe allows you to bend the rules of the language, so be careful; as with any warning, the compiler may not generate what you expect.

Note that there is no way to specify recovery from code-F or -I errors; these are always fatal and prevent generation of a compiled output file.

1.6.2 Suppressing Warning Messages (–pw Option)

The –pw option enables you to suppress the output of warning messages, causing the compiler to quietly ignore code-W messages (warnings). This is useful when you are aware of the condition causing the warning and consider it innocuous.

You can double –pw options to suppress code-E errors as well: doubling the –pw option suppresses both code-W and code-E errors. Doubling –pw is useful in conjunction with the –pe option so that code-E errors are ignored completely. For example:

```
ppcl –peww *.c  ; completely ignore all W and E errors
```

1.6.3 An Example of How You Can Use Error Options

The following example demonstrates how the –pe and –pw options can be used to suppress errors and error messages. The examples use this 2-line code segment:

```
int *pi; char *pc;
pi = pc;
```

- If you invoke the code with the mpcl or ppcl shell (and –q), this is the result:

```
[err]
"err.c", line3: [E104] operands of '=' point to different types
```

In this case, because code-E errors are fatal, the compiler does not generate code.

- If you invoke the code with the mpcl or ppcl shell and the –pe option, this is the result:

```
[err]
"err.c", line3: [E104] operands of '=' point to different types
```

In this case, the same message is generated, but because –pe is used, the compiler ignores the error and generates an output file.

- If you invoke the code with the mpcl or ppcl shell and –peww (–pe and double –pw), this is the result:

```
[err]
```

As in the previous case, –pe causes the compiler to overlook the error and generate code. Because the two –pw options are used, the message is suppressed.

1.7 Linking C Code

The TMS320C8x C compilers and assembly language tools provide two methods for linking your programs:

- You can compile and assemble individual modules and then link them together. This is especially useful when you have multiple source files.
- You can compile, assemble, and link in one step by using the mpcl or ppcl shell. This is useful when you have a single source module.

This section describes how to invoke the linker with each method. It also discusses special requirements of linking C code, including using the runtime-support libraries, specifying the initialization model, and allocating the program into memory.

1.7.1 Invoking the Linker

The TMS320C8x C compilers and assembly language tools support modular programming by allowing you to compile and assemble individual modules and then link them together. This is the general syntax for linking C programs in a separate step:

mvplnk {**–c**|**–pc**|**–cr**} *filenames* [*options*] **–o** *name.out* **–l** *libraryname*

mvplnk	is the command that invokes the linker.
–c|**–pc**|**–cr**	are options that tell the linker to use special conventions that are defined by the C environment. Note that when you use the mpcl shell to link, the shell automatically passes –c to the linker; when you use the ppcl shell to link, the shell automatically passes –pc to the linker.
filenames	are object files created by compiling and assembling programs.
options	are other options that affect the way the linker processes input files. They are listed in the Linker Options section of Table 1–1, *Option Summary Table*.
–o *name.out*	names the output file. If you don't use the –o option, the linker creates an output file with the default name of **a.out**.

–l*libraryname* identifies the appropriate archive library containing C runtime-support functions. Four runtime-support libraries, named mp_rts.lib, mp_rtsl.lib, pp_rts.lib, and pp_rtsl.lib, are included with the C compiler. If you are linking C code, you must use these libraries or one that you created from mp_rts.src or pp_rts.src using the library build utility.

Whenever you specify a library as linker input, the linker includes and links only those library members that resolve undefined references. The mp_rts.lib library is used to resolve references from big endian MP C code, and pp_rts.lib is used to resolve references from big endian PP C code. For example, you can link a C program consisting of modules prog1, prog2, and prog3 (the output file is named prog.out):

```
mvplnk –c prog1 prog2 prog3 –l mp_rts.lib –o prog.out
```

If you use any C input/output functions, you must also link in the appropriate CIO library. The CIO libraries are provided with the C compilers. For a complete description of these libraries, see Chapter 5, *Runtime-Support Functions*.

The linker uses a default allocation algorithm to allocate your program into memory. You can use the MEMORY and SECTIONS linker directives to customize the allocation process. These directives are described in Section 13.6, *The MEMORY Directive*. and Section 13.7, *The SECTIONS Directive*.

1.7.2 Using the Shell to Invoke the Linker (–z Option)

The options and parameters discussed in this section apply to both methods of linking; however, when you are linking with the shell, the options follow the –z shell option.

By default, the shell does not run the linker. However, if you use the **–z** option, the shell compiles, assembles, and links in one step. When using –z to enable linking, remember that:

- –z must follow all source files and compiler options on the command line (or be specified with C_OPTION).
- –z divides the command line into compiler options (before –z) and linker options (following –z).
- –c and –n suppress –z, so do not use them if you want to link.

All arguments that follow –z on the command line are passed to the linker. These arguments can be linker command files, additional object files, linker options, or libraries.

For example, to compile and link all the .c files in a directory, enter:

```
ppcl –q *.c –z pplnk.cmd –o pp.out
```

The shell compiles all of the files in the current directory that have a .c extension with the –q option. When the compiler is finished, it invokes the linker (–z option). The linker, using options specified in the pplnk.cmd command file, will link together all object files produced by the compiler run, the runtime support library, and any additional files specified in pplnk.cmd. The resulting linked file will be called pp.out.

The order in which the linker processes arguments can be important, especially for command files and libraries. The shell passes arguments to the linker in the following order:

1) Object filenames passed from by the compiler
2) Arguments following –z on the command line
3) Arguments following –z from the C_OPTION environment variable

The –c shell option

You can override the –z option by using the –c mpcl or ppcl shell option. This option is especially helpful when you have specified –z in the C_OPTION environment variable and want to selectively disable linking with –c on the command line.

The –c linker option has a different function than, and is independent of, the –c shell option.

The –n shell option

The –n shell option also overrides the –z option. The –n shell option is a specialized tool that compiles only and does not assemble or link the files.

1.7.3 Controlling the Linking Process

Regardless of the method you choose for invoking the linker, special requirements apply when linking C programs. You must:

- ❑ Include the compiler's runtime-support library
- ❑ Specify the initialization model
- ❑ Determine how you want to allocate your program into memory

This section discusses how these factors are controlled and provides an example of the standard default linker command file.

For more information about how to operate the linker, see Chapter 13, *Linker Description.*

Runtime-support libraries

All C programs must be linked with a runtime-support library. This archive library contains standard C library functions (such as malloc and strcpy) as well as functions used by the compiler to manage the C environment. To link in a library, simply use the –l option on the command line:

```
mvplnk –pc  filenames     –l  pp_rts.lib ...
   or
mvplnk –c  filenames      –l  mp_rts.lib ...
```

Generally, the libraries should be specified last on the command line because the linker searches libraries for unresolved references in the order that files are specified on the command line. If any object files follow a library, references from those object files to that library will not be resolved. You can use the –x option to force the linker to reread all libraries until references are resolved. Whenever you specify a library as linker input, the linker includes and links only those library members that resolve undefined references.

All C programs must be linked with an object module called *boot.obj*. When a program begins running, it executes boot.obj first. boot.obj contains code and data for initializing the runtime environment; the linker automatically extracts boot.obj and links it when you use –c, –cr, or –pc options and include one of the TMS320C8x runtime-support libraries (such as mp_rts.lib) in the link.

If a C program uses any of the input/output facilities provided by the CIO library, the CIO library must also be linked in. See Chapter 5, *Runtime-Support Functions*, for a complete description of the runtime-support and C I/O libraries.

Chapter 5, *Runtime-Support Functions*, describes additional runtime-support functions that are included in the runtime-support and C I/O libraries. These functions include ANSI C standard runtime support and access to MVP specific operations and instructions.

Initialization (RAM and ROM models)

The C compiler produces tables of data for autoinitializing global variables. Subsection 3.8.1, *PP Autoinitialization of Variables and Constants*, and subsection 4.8.1, *MP Autoinitialization of Variables and Constants*, discuss the format of these tables. These tables are in a named section called *.cinit* (MP) or *.pcinit* (PP). The initialization tables can be used in either of two ways:

- **RAM model** (–cr linker option)

  Global variables are initialized at **load time**; use the –cr (MP) linker option to select the RAM model. For more information about the RAM model, see *Initializing variables in the RAM model for the MP*, page CG:4-31.

- **ROM model** (–c or –pc linker option)

  Global variables are initialized at **runtime**; use the –c (MP) or –pc (PP) linker option to select the ROM model. For more information about the ROM model, see *ROM initialization model for the PP*, page CG:3-31 or *ROM initialization model for the MP*, page CG:4-32.

The –c, –pc, and –cr linker options

Whenever you link a C program, you must use either the –c, –pc, or the –cr option. These options tell the linker to use special conventions required by the C environment; for example, they tell the linker to select the ROM or RAM model of autoinitialization, and to link for either MP or PP programs. Note that when you use the mpcl shell to link programs, the –c option is the default; when you use the ppcl shell to link programs, the –pc option is the default. The following list outlines the linking conventions used with –c, –pc, or –cr:

- The symbol **_c_int00** or **$c_int00** is defined as the program entry point; it identifies the beginning of the C boot routine in boot.obj. When you use –c, or –cr, _c_int00 is automatically referenced; this ensures that boot.obj is automatically linked in from mp_rts.lib. When you use –pc, $c_int00 is automatically referenced; this ensures that boot.obj is automatically linked in from pp_rts.lib.

- The **.cinit** (–c or –cr) or **.pcinit** (–pc) output section is padded with a termination record so that the loader (RAM model) or the boot routine (ROM model) knows when to stop reading the initialization tables.

- In the **RAM model** (–cr option)
  - The linker sets the symbol cinit to –1. This indicates that the initialization tables are not in memory, so no initialization is performed at runtime.
  - The STYP_COPY flag (010h) is set in the .cinit section header. STYP_COPY is a special attribute that tells the loader to perform autoinitialization directly and not to load the .cinit or section into memory. The linker does not allocate space in memory for the .cinit section. *Note that a loader is not included as part of the C compiler package.*
- In the **ROM model** (–c or –pc option), the linker defines the symbol cinit (–c) or $pcinit (–pc) as the starting address of the .cinit or .pcinit section. The C boot routine uses one of these symbols as the starting point for autoinitialization.

Sections created by the compiler

The compiler produces several relocatable blocks of code and data. These blocks, called *sections*, can be allocated into memory in a variety of ways to conform to a variety of system configurations.

The compiler creates two basic kinds of sections: initialized and uninitialized. Table 1–2 summarizes the sections.

Table 1–2. Sections Created by the Compiler

Name	Type	Contents
.bss	Uninitialized	global and static variables (MP) and global and static variables created with *far* keyword (PP)
.pbss	Uninitialized	global and static variables (PP)
.cinit	Initialized	tables for explicitly initialized global and static variables (MP)
.pcinit	Initialized	tables for explicitly initialized global and static variables (PP)
.const	Initialized	string literals and floating-point constants
.switch	Initialized	switch tables
.stack	Uninitialized	stack (MP)
.pstack	Uninitialized	stack (PP)
.text	Initialized	executable code (MP)
.ptext	Initialized	executable code (PP)
.sysmem	Uninitialized	memory for malloc functions (MP)
.psysmem	Uninitialized	memory for malloc functions (PP)

When you link your program, you must specify where to locate the sections in memory. In general, initialized sections can be linked into ROM or RAM; uninitialized sections must be linked into RAM. Refer to subsection 3.1.1, *PP Sections,* or subsection 4.1.1, *MP Sections*, for a complete description of how the compiler uses these sections. The linker provides MEMORY and SECTIONS directives for performing this process.

For more information about allocating sections into memory, see Chapter 13, *Linker Description.*

Sizing the stack and heap

The linker provides options that allow you to specify the size of the .stack, .pstack, .sysmem, and .psysmem sections.

–stack*size*	sets the size of the MP stack, by setting the size of the .stack section to *size* bytes. The value *size* must be constant.
–pstack*size*	sets the size of the PP stack, by setting the size of the .pstack section to *size* bytes. The value *size* must be constant.
–heap*size*	sets the size of the MP heap by setting the size of the .sysmem section to *size* bytes. The value *size* must be constant.
–pheap*size*	sets the size of the PP heap, by setting the size of the .psysmem section to *size* bytes. The value *size* must be constant.

The .sysmem and .psysmem sections exist only if you use memory allocation functions (such as malloc()). The linker resizes these sections only if the value specified by the option is larger than the input section size (in the standard library, the size is 0, so any –stack, –pstack, –heap, or –pheap option takes effect). The default size for the .pstack and .psysmem sections is 128 bytes. The default size for the .stack and .sysmem sections is 1028 bytes.

Sample linker command file

The compiler package includes sample linker command files called pplnk.cmd for the PP and mplnk.cmp for the MP, which can be used to link C programs. To link your program using a command file, enter one of the following commands:

```
mvplnk object.files –o output.file –m map.file pplnk.cmd
mvplnk object.files –o output.file –m map.file mplnk.cmd
```

Figure 1–3 shows the contents of pplnk.cmd.

Figure 1–3. Sample Linker Command File for the PP

```
-pc
-x
-pheap  0x800
-pstack 0x580
-u $exit
-l pp_cio.lib
-l pp_rts.lib

MEMORY
{
    DRAM01    : o=0x00000004   l = 0x03ffc
    DRAM2     : o=0x00008000   l = 0x00800
    PRAM0     : o=0x01000200   l = 0x00600
    EXTMEM    : o=0x02000000   l = 0x80000
}

SECTIONS
{
    .ptext   :  > EXTMEM
    .const   :  > EXTMEM
    .switch  :  > EXTMEM
    .pcinit  :  > EXTMEM
    .pbss    :  (PASS) > DRAM01
    .psysmem :  (PASS) > DRAM2
    .pstack  :  (PASS) > PRAM0
}
```

In this command file:

- The –pc option specifies that the linker use the ROM initialization model for PP C code. See Section 3.8, *System Initialization* for a complete description of the ROM initialization model.
- The –x option tells the linker to repeatedly search object libraries to resolve back referenced symbols. See subsection 13.3.18, *Exhaustively Read Libraries (–x Option)*.
- The –pheap and –pstack options are used to specify sizes for the .psysmem and .pstack sections.
- The –l (lowercase L) option with pp_rts.lib and pp_cio.lib specifies that the linker search for those libraries. If the libraries are not in the current directory, you can customize the command file to use C_DIR or the –i option to define a path where the libraries can be found.

- The –u $exit option is used when linking in both the pp_rts.lib and pp_cio.lib. Both libraries contain a definition of the symbol $exit. Using the –u $exit option forces the version of the symbol from the pp_cio.lib library to be linked in. If your C program does not reference any C input/output functions, then the –u $exit and –l pp_cio.lib options are not necessary. See Chapter 5, *Runtime-Support Functions*, for a description of these libraries and how to link them with your C code.

- The MEMORY and SECTIONS commands are used to describe the memory map and memory allocation scheme to be implemented by the linker.

You will most likely have to customize the command file to fit a particular application by adding or modifying options, libraries, memory configurations, and section allocations.

For more information about operating the linker, see Chapter 13, *Linker Description.*

Chapter 2

TMS320C8x C Language

The C language that the TMS320C8x C compilers support is based on the ANSI (American National Standards Institute) C standard. The TMS320C8x C compilers strictly conform to the ANSI C standard. The ANSI standard identifies certain implementation-defined features that may differ from compiler to compiler. This chapter describes how these and other features are implemented for the TMS320C8x C compilers.

Topics

2.1 Characteristics of TMS320C8x C

The ANSI standard identifies some features of the C language that are affected by characteristics of the target processor, runtime environment, or host environment. For reasons of efficiency or practicality, this set of features may differ among standard compilers. This section describes how these features are implemented for the TMS320C8x C compiler.

The following list identifies all such cases and describes the behavior of the TMS320C8x C compiler in each case. Each description also includes a reference to the formal ANSI standard and to the *The C Programming Language* by Kernighan and Ritchie (K&R).

Nonstandard keywords

The following keywords have been added to the TMS320C8x C language and are not defined in the ANSI standard:

- ❑ inline — see subsection 1.4.2, *Definition-Controlled Inline Function Expansion*
- ❑ asm — see Section 2.9, *The asm Statement*
- ❑ cregister — see Section 2.5, *The cregister Keyword*
- ❑ interrupt — see Section 2.6, *The interrupt and trap Keywords*
- ❑ shared — see Section 2.7, *The shared and sharedpp Keywords*
- ❑ sharedpp — (PP only) see Section 2.7, *The shared and sharedpp Keywords*
- ❑ trap — (MP only) see Section 2.6, *The interrupt and trap Keywords*
- ❑ far — (PP only) see Section 2.4, *The far Keyword (PP only)*

Identifiers and constants

- ❑ The first 31 characters of all identifiers are significant. Case is significant; uppercase and lowercase characters are distinct for identifiers. These characteristics apply to all identifiers, internal and external, in all TMS320C8x tools. (ANSI 3.1.2, K&R A2.3)
- ❑ The source (host) and execution (target) character sets are assumed to be ASCII. Although the compiler recognizes the syntax of multibyte characters, there are no additional multibyte characters. (ANSI 2.2.1, K&R A12.1)

- Hex or octal escape sequences in character or string constants may have values up to 32 bits. (ANSI 3.1.3.4, K&R A2.5.2)
- Character constants with multiple characters are encoded as the last character in the sequence. For example,
  `'abc' == 'c'` (ANSI 3.1.3.4, K&R A2.5.2)

Data types

For information about the representation of data types, see Section 2.2, *Data Types*. (ANSI 3.1.2.5, K&R A4.2)

- The type size_t, which is assigned to the result of the sizeof operator, is equivalent to unsigned int. (ANSI 3.3.3.4, K&R A7.4.8)
- The type ptrdiff_t, which is assigned to the result of pointer subtraction, is equivalent to int. (ANSI 3.3.6, K&R A7.7)

Conversions

- Int-to-float conversions on the MP use the frndz instruction which rounds toward 0. These conversions are simulated on the PP and they also round toward 0. (ANSI 3.2.1.3, K&R A6.3)
- Pointers and integers can be freely converted. (ANSI 3.3.4, K&R A6.6)

Expressions

- When two signed integers are divided and either is negative, the quotient is negative. A signed modulus operation takes the sign of the dividend (the first operand). For example,

  ```
  10 / –3 == –3,    –10 / 3 == –3
  10 % –3 == 1,     –10 % 3 == –1
  ```
  (ANSI 3.3.5, K&R A7.6)
- A right-shift of a signed value is an arithmetic shift; that is, the sign is preserved. (ANSI 3.3.7, K&R A7.8)

Declarations

- The *register* storage class is effective for all character, short, integer, and pointer types. (ANSI 3.5.1, K&R A8.1)
- Structure members are packed into words. (ANSI 3.5.2.1, K&R A8.3)

- A bit field of type integer is signed. Bit fields are packed into words, and do not cross word boundaries. For more information on bit field packing algorithms see Section 3.2.2, *PP Bit Fields*, and see Section 4.3.2, *MP Bit Fields*.
  (ANSI 3.5.2.1, K&R A8.3)

- The cregister keyword is only effective for the register names listed in Section 2.5, *The cregister Keyword*, and can only be applied to file scope level variables with integral types.
  (T.I. C extension)

- The interrupt or trap (MP) keyword can only be applied to void functions with no arguments. (T.I. C extension)

- The shared or sharedpp keywords can only be applied to file scope level variables. (T.I. C extension)

Preprocessor

- The preprocessor recognizes four #pragma directives; all others are ignored. For details, see Section 2.8, *Pragma Directives*. The recognized pragmas are:

  #pragma DATA_ALIGN (*symbol* **,** *constant* **)**
  #pragma DATA_SECTION (*symbol* **, "** *section name***")**
  #pragma SHARED (*symbol***)**
  #pragma SHAREDPP (*symbol***)**
  (ANSI 3.8.6, K&R A12.8)

2.2 Data Types

- The char, unsigned char, and signed char data types are represented as 8-bit values.
- The short and unsigned short data types are represented as 16-bit values.
- The int, unsigned int, long, and unsigned long data types are represented as 32-bit values.
- All signed data types are represented in 2s-complement notation.
- The char data type is equivalent to the signed char data type.
- The enum data type is represented as a 32-bit value. In expressions, the enum type is equivalent to the int type.
- For the PP, all floating-point data types (float, double, and long double) are equivalent and represented in IEEE 32-bit floating-point format.
- For the MP, the float data type is represented in IEEE 32-bit floating-point format. The double and long double data types are equivalent and are represented in IEEE 64-bit floating-point format.

The size, representation, and range of each scalar data type are listed in Table 2–1. Many of the range values are available as standard macros in the header file limits.h, which is supplied with the compiler.

Table 2–1. TMS320C8x C Data Types

Type	Size	Representation	Minimum	Maximum
char, signed char	8 bits	ASCII	–128	127
unsigned char	8 bits	ASCII	0	255
short	16 bits	2s complement	–32768	32767
unsigned char	16 bits	binary	0	65 535
int, signed int	32 bits	2s complement	–2147483648	2147483647
unsigned int	32 bits	binary	0	4294967295
long, signed long	32 bits	2s complement	–2147483648	2147483647
unsigned long	32 bits	binary	0	4294967295
enum	32 bits	2s complement	–2147483648	2147483647
float	32 bits	IEEE 32-bit	1.175494e–38	3.40282346e+38
double (PP)	32 bits	IEEE 32-bit	1.175494e–38	3.40282346e+38
long double (PP)	32 bits	IEEE 32-bit	1.175494e–38	3.40282346e+38
double (MP)	64 bits	IEEE 64-bit	2.22507385e–308	1.79769313e+308
long double (MP)	64 bits	IEEE 64-bit	2.22507385e–308	1.79769313e+308
pointers	32 bits	binary	0	0xFFFFFFFF

2.3 Register Variables

The TMS320C8x C compilers treat register variables (variables declared with the *register* keyword) differently, depending on whether you use the optimizer.

Compiling with the optimizer

The compiler ignores any register declarations and allocates registers to variables and temporary values by using a cost algorithm that attempts to maximize register use.

Compiling without the optimizer

If you use the register keyword, you can suggest variables as candidates for allocation into registers. The same set of registers used for allocation to temporary expression results is used for allocation of register variables. For more information about PP register variables, see Section 3.3, *PP Register Conventions*, and for MP register variables, see Section 4.4, *MP Register Conventions*.

The compiler will attempt to honor all register declarations; however, spilling may occur if the compiler runs out of appropriate registers. Spilling is the process of moving a register's contents to memory to free the register. Declaring too many objects as register variables could limit the number of registers the compiler can use to allocate for temporary expression results and could cause excess spilling. Use the register keyword carefully.

Any object with a scalar type (integer, floating-point, or pointer) can be declared as a register variable. The register designator is ignored for objects of other types.

The register storage class is meaningful for parameters as well as local variables. Normally, in a function, some of the parameters are copied to a location on the stack where they are referenced during the function body. A register parameter will be copied to a register instead of the stack. This will speed access to the parameter within the function. For more information about which parameters are normally passed in registers, see Section 4.5 (MP) or 3.4 (PP), *Function Structure and Calling Conventions.*

2.4 The far Keyword (PP Only)

The TMS320C8x PP C compiler extends the C language with the far keyword to specify storage allocation for global and static variables. By default, if a global or static object is not declared as far it will be allocated in the .pbss section, which is assumed to be linked into on-chip data or parameter RAM. Since the compiler assumes these objects are on-chip, it generates code using the PP relative addressing mode to reference them. For example, for this declaration

```
extern int i;
```

the object i would be referenced as *(xba + $i). Since the PP relative addressing mode has a limited range (15 bits scaled), it is not useful for referencing objects linked into external memory (which have large address values).

The far keyword was added to the PP C language to indicate that an object should be referenced with a full 32-bit address. This allows PP objects to be allocated in external memory and PP C code to reference external objects created by the MP. When the far keyword is used in the definition of an object, the object will be allocated in the .bss section instead of the .pbss section. The compiler assumes that you will link the .bss section into external memory. For more information, see subsection 3.1.3, *Static and Global Memory Models*.

The far keyword is treated as a storage class modifier. It can appear anywhere in a declaration—before, after, or in between the storage class specifiers and types. For example:

```
far static int x;
static far int x;
static int far x;
```

Storage class modifiers are meaningful only for object declarations, not for functions; also, two storage class modifiers cannot be used together in a single declaration. The far keyword can be used for any declaration:

- Containing a static storage class keyword
- Containing an extern storage class keyword
- In file scope (an external definition) that has no class specifier

Example 2–1. The far Keyword

C source file

```
int     plain;
far int ext;

main()
{
    extern far int farvar;
    extern int     flev;

    farvar = plain;
    flev   = ext;
}
```

Assembly source file

```
$main:
          .global $farvar
          .global $flev
;>>>>      farvar = plain;
          a0 = $farvar
          d1 =sw  *(xba + $plain)
          *a0 =w  d1
;>>>>      flev   = ext;
          a0 = $ext
           nop
          d1 =sw  *a0
          *(xba + $flev) =w  d1
          br = iprs
           nop
           nop
;         branch occurs here
          .global $ext
          .bss $ext,4,4
          .global $plain
$plain:   .usect  .pbss,4,4
```

2.5 The cregister Keyword

The MVP compilers extend the C language by adding the cregister keyword to allow high level language access to control registers.

When you use the cregister keyword on an object, the compiler compares the name of the object to a list of standard control registers for the architecture you are compiling for (see Table 2–2 and Table 2–3). If the name matches, the compiler will generate the code to reference the control register. If the name does not match, the compiler will issue an error.

In the PP, control registers can be treated as normal registers, so the compiler will generate normal code. For the MP, however, additional code is generated because the rdcr instruction is required to read the control register into a normal register, and the wrcr instruction writes the data back to the control register.

Table 2–2. Valid Control Registers for the PP

COMM	INTFLG	INTEN	MF
TAG0	TAG1	TAG2	TAG3

Table 2–3. Valid Control Registers for the MP

ANASTAT	BRK1	BRK2	CONFIG	DLRU
DTAG0	DTAG1	DTAG2	DTAG3	DTAG4
DTAG5	DTAG6	DTAG7	DTAG8	DTAG9
DTAG10	DTAG11	DTAG12	DTAG13	DTAG14
DTAG15	ECOMCNTL	EIP	EPC	FLTADR
FLTDTH	FLTDTL	FLTOP	FLTTAG	FPST
IE	ILRU	IN0P	IN1P	INTPEN
ITAG0	ITAG1	ITAG2	ITAG3	ITAG4
ITAG5	ITAG6	ITAG7	ITAG8	ITAG9
ITAG10	ITAG11	ITAG12	ITAG13	ITAG14
ITAG15	MIP	MPC	OUTP	PKTREQ
PPERROR	SYSSTK	SYSTMP	TCOUNT	TSCALE

The control register names are case sensitive; they should be specified in all capital letters.

The cregister keyword is treated as a storage class modifier. It can appear anywhere in a declaration; before, after, or in between the storage class specifiers and types. Storage class modifiers are meaningful only for object declarations, not for functions. Two storage class modifiers cannot be used together in a single declaration.

The cregister keyword can be used for any declaration in file scope. The cregister keyword is not allowed on any declaration within the boundaries of a function. It can only be used on objects of type integer or pointer. The cregister keyword is not allowed on objects of any floating-point type or on any structure or union objects.

The cregister keyword does not imply that the object is volatile. If the control register being referenced is volatile (that is, can be modified by some external control), then the object should be declared with the volatile keyword also.

The header file mvp.h, shipped with the compilers, declares all the cregister names visible to the PP and MP compiler. You can include this header file by specifying:

```
#include <mvp.h>
```

and then use the names from Table 2–2 and Table 2–3 directly. Refer to Section 5.5, *Runtime-Support Functions* for more information on mvp.h.

2.6 The interrupt and trap Keywords

The MVP C compilers extend the C language by adding the interrupt keyword to specify that a function is to be treated as an interrupt function. The MP also adds the trap keyword for software traps.

Functions that handle interrupts require special register saving rules and a special return sequence. When you use the interrupt keyword on the definition of the function, the compiler generates register saves based on the rules for interrupt functions and the special return sequence for interrupts. For more information on interrupt functions, see subsection 3.6.2, *Using PP C Interrupt Routines*, or subsection 4.7.2, *Using MP C Interrupt Routines*,

The interrupt keyword can only be used on a function that is defined to return void, and that has no parameters. The body of the interrupt function can have local variables, and is free to use the stack. For example,

```
interrupt void int_handler()
{
    unsigned int flags;

    ....
}
```

The trap keyword is only supported in the MP Compiler, and is identical to the interrupt keyword, except for the return sequence that is generated by the compiler. The trap keyword is necessary for the MP, since a software trap for the MP requires a slightly different return sequence than a hardware interrupt.

2.7 The shared and sharedpp Keywords

The MVP compilers extend the C language by adding the shared and sharedpp keywords.

The shared keyword

The shared keyword allows C symbol names to be visible to both MP and PP C source code. For example, a symbol defined in a PP module using the shared keyword can be referenced by an MP module when the two modules are linked together. The shared keyword also makes a symbol visible among separate PP tasks that are linked together.

The shared keyword can be used for any declaration in file scope. It is not allowed on any declaration within the boundaries of a function.

The shared keyword is treated as a storage class modifier that is, it can appear anywhere in a declaration; before, after, or in between the storage class specifiers and types. For example:

```
shared int var;
extern shared void pp_entry_point();
```

The shared keyword cannot be used with other storage class modifiers.

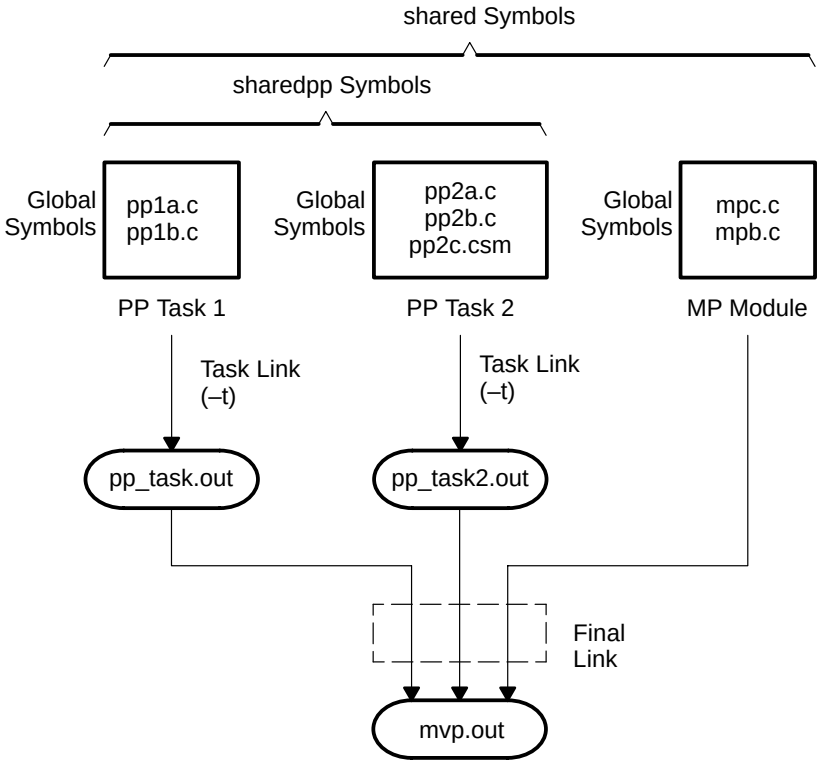
The sharedpp keyword (PP only)

The sharedpp keyword behaves the same as the shared keyword except that sharedpp only makes symbols global among PP tasks. Unlike the shared keyword, sharedpp does not make a symbol visible to MP modules.

A PP task is a collection of source code modules that have been compiled and linked together and is meant to run on one PP. Normally, when a PP task is linked the global symbols defined in the task are specific to that task. A global symbol defined in one PP task cannot be referenced in another PP task when they are linked together. A global symbol declared with the sharedpp keyword becomes visible to all PP tasks.

Figure 2–1 illustrates the levels of visibility provided by the shared and sharedpp keywords. PP task 1 is made up of modules pp1a.c and pp1b.c. Global symbols defined in pp1a.c and pp1b.c are visible to each other. Global symbols declared in pp1a.c and pp1b.c that use the sharedpp keyword are also visible to modules in PP task 2. Symbols declared with the shared keyword are also visible to MP modules mp1.c and mp2.c.

Figure 2–1. The sharedpp Keyword



The function of the shared and sharedpp keywords can also be achieved with pragmas. For more information about pragmas, see Section 2.8, *Pragma Directives*. For more information on how to used the shared and sharedpp keywords when linking MP and PP code and when linking PP tasks see subsection 13.3.15, *Task Level Linking (–t Option)*, and Chapter 14, *Linking PP and MP Files: An Extended Example*.

2.8 Pragma Directives

The MVP C compilers support the following pragmas.

#pragma DATA_ALIGN (*symbol***,** *constant***)**
#pragma DATA_SECTION (*symbol***, "***section name***")**
#pragma SHARED (*symbol***)**
#pragma SHAREDPP (*symbol***)**

The *symbol* must have file scope; that is, it cannot be defined or declared inside the body of a function. The pragma itself must be specified outside the body of a function, and must occur before any declaration, definition, or reference to the symbol. If you do not follow these rules, the compiler may issue a warning or silently ignore the pragma.

The DATA_ALIGN pragma

The DATA_ALIGN pragma will align the *symbol* to an alignment boundary. The alignment boundary is the maximum of the symbol's default alignment value or the value of the *constant* in bytes. The constant must be a power of 2.

The DATA_SECTION pragma

The DATA_SECTION pragma will allocate space for the *symbol* in a section named *section name*. The *section name* is limited to a maximum of eight characters.

The DATA_SECTION pragma is useful if you have data objects that you would like to link into an area separate from the .bss or .pbss. This is especially useful on the PP when you have two buffers and you want them to be in different data RAM banks. Using the DATA_SECTION pragma, you could specify that one of the buffers should go in another section, and then you could link that section into a different data RAM with the linker.

Example 2–2. Using the DATA_SECTION Pragma

MP C source file

```
#pragma DATA_SECTION(bufferB, "my_sect")
char bufferA[512];
char bufferB[512];
```

Assembly source file

```
    .global   _bufferA
    .bss   _bufferA,512,1

    .global   _bufferB
_bufferB: .usect "my_sect",512,1
```

The SHARED pragma

The SHARED pragma allows C symbol names to be visible to both MP and PP C source code. The SHARED pragma behaves like the shared keyword. For more information on the shared keyword, see Section 2.7, *The shared and sharedpp Keywords*.

The SHAREDPP pragma

The SHAREDPP pragma behaves the same as the SHARED pragma except that SHAREDPP makes symbols global only among PP tasks. Unlike the SHARED pragma, SHAREDPP does not make a symbol visible to MP modules. The SHAREDPP pragma behaves like the sharedpp keyword. For more information on the sharedpp keyword, see Section 2.7, *The shared and sharedpp Keywords*.

2.9 The asm Statement

The TMS320C8x C compilers allow you to imbed TMS320C8x MP or PP assembly language instructions or directives directly into the assembly language output of the compilers. This capability is provided through an extension to the C language: the *asm* statement. The asm statement is syntactically like a call to a function named asm, with one string-constant argument:

```
asm("assembler  text");
```

The compiler copies the argument string directly into your output file. The assembler text must be enclosed in double quotes. All the usual character string escape codes retain their definitions. For example, you can insert a .string directive that contains quotes:

```
asm("STR: .string \"abc\"");
```

The inserted code must be a legal assembly language statement. Like all assembly language statements, the line *must* begin with a label, a blank, a tab, or a comment (asterisk or semicolon). The compiler performs no checking on the string; if there is an error, it will be detected by the assembler. For more information, see Chapter 7, *Assembler Description*.

asm statements do not follow the syntactic restrictions of normal C statements: they can appear as either a statement or a declaration, even outside blocks. This is particularly useful for inserting directives at the beginning of a compiled module.

Note: Avoid Disrupting the C Environment With asm Statements

Be extremely careful not to disrupt the C environment with asm statements. The compiler does not check the inserted instructions. Inserting jumps and labels into C code can cause unpredictable results in variables manipulated in or around the inserted code. Directives that change sections or otherwise affect the assembly environment can also be troublesome. Be especially careful when you use the optimizer with asm statements. Although the optimizer cannot remove asm statements, it can significantly rearrange the code order near asm statements, possibly causing undesired results. The asm command is provided so that you can access features of the hardware, which, by definition, C is unable to access.

For more information on the PP runtime environment, see Chapter 3, *PP Runtime Environment*, and for more information on the MP runtime environment see Chapter 4, *MP Runtime Environment*.

2.10 Initializing Static and Global Variables

The ANSI C standard specifies that static and global variables without explicit initializations must be preinitialized to zero (before the program begins running). This task is typically done when the program is loaded. Because the loading process depends heavily on the specific environment of the target application system, the TMS320C8x C compiler itself makes no provision for preinitializing variables; therefore, it is up to your application to fulfill this requirement.

2.11 Compatibility With K&R C

The ANSI C language is a superset of the de facto C standard defined in the first edition of *The C Programming Language* by Kernighan and Ritchie (K&R). Most programs written for earlier non-ANSI compilers should correctly compile and run without modification.

However, there are subtle changes in the language that may affect existing code. Appendix C in K&R (second edition) summarizes the differences between ANSI C and the previous C standard (first edition, hereafter referred to as K&R C).

To simplify the process of compiling existing C programs with the TMS320C8x ANSI C compilers, the compilers have a K&R option (–pk) that modifies some of the semantic rules of the language for compatibility with older code. In general, the –pk option relaxes requirements that are stricter for ANSI C than for K&R C. The –pk option does not disable any new features of the language such as function prototypes, enumerations, initializations, or preprocessor constructs. Instead, –pk simply liberalizes the ANSI rules without revoking any of the features.

The specific differences between ANSI C and K&R C are as follows:

- The integral promotion rules have changed regarding promoting an unsigned type to a wider signed type. Under K&R, the result type was an unsigned version of the wider type; under ANSI, the result type is a signed version of the wider type. This affects operations that perform differently when applied to signed or unsigned operands—namely, comparisons, division (and mod), and right shift. In this example, assume that short is narrower than int:

```
unsigned short u;
int i;
if (u < i) ... /* SIGNED comparison, unless –pk used */
```

- ANSI prohibits two pointers to different types from being combined in an operation. In most K&R compilers, this situation is only a warning. Such cases are still diagnosed when –pk is used, but with lower severity:

```
int *p;
char *q = p;  /* error without –pk, warning with –pk */
```

  Even without –pk, a violation of this rule is a code-E (recoverable) error. –pe, which converts code-E errors to warnings, can be used as an alternative to –pk.

- External declarations with no type or storage class (only an identifier) are illegal in ANSI but legal in K&R:

```
a;              /* illegal unless –pk used */
```

- ANSI interprets file scope definitions that have no initializer as *tentative definitions*: in a single module, multiple definitions of this form are fused together into a single definition. Under K&R, each definition is treated separately. For example,

```
int a;
int a;              /* illegal if –pk used, OK if not */
```

- ANSI prohibits, but K&R allows, objects with external linkage to be redeclared as static:

```
extern int a;
static int a;       /* illegal unless –pk used */
```

- Unrecognized escape sequences in string and character constants are explicitly illegal under ANSI but ignored under K&R:

```
char c = '\q';      /* same as 'q' if –pk used, error
                     if not */
```

- ANSI specifies that bit fields must be of type int or unsigned. With the K&R –pk option, bit fields can be legally declared with any integral type. For example,

```
struct s
{
 short f : 2;       /* illegal unless –pk used */
};
```

- ANSI syntax prohibits, but K&R syntax allows, a trailing comma in enumerations:

```
enum { a, b, c, }; /* illegal unless –pk used */
```

- ANSI syntax prohibits, but K&R syntax allows, trailing tokens on preprocessor directives:

```
#endif NAME         /* illegal unless –pk used */
```

- ANSI syntax prohibits, but K&R syntax allows, assignments to structures returned from a function:

```
f().x = 123         /* illegal unless –pk used */
```

2.12 Compiler Limits

The TMS320C8x compilers impose some compile time limits that restrict them from compiling files that are excessively large or complex. Most of these conditions occur during the first compilation pass (parsing). When such a condition occurs, the parser issues a code-I diagnostic message indicating the condition that caused the failure; usually the message also specifies the maximum value for whatever limit has been exceeded. The code generator also has compilation limits, but it has fewer than the parser has.

In general, exceeding any compiler limit prevents continued compilation, so the compiler aborts immediately after printing the error message. The only way to avoid exceeding a compiler limit is to simplify the program or parse and preprocess in separate steps.

Many compiler tables have no absolute limits but rather are limited only by the amount of memory available on the host system. Table 2–4 specifies the limits that are absolute. All the absolute limits equal or exceed those required by the ANSI C standard.

Table 2–4. Absolute Compiler Limits

Description	Limits
Filename length	512 characters
Source line length	16K characters (Note 1)
Length of strings built from # or ##	512 characters (Note 2)
Inline assembly string length	132 characters
Macros predefined with –d	64
Macro parameters	32 parms
Macro nesting level	64 levels (Note 3)
#include search paths	64 paths (Note 4)
#include file nesting	64 levels
Conditional inclusion (#if) nesting	64 levels
Nesting of struct, union, or prototype declarations	20 levels
Function parameters	48 parms
Array, function, or pointer derivations on a type	12 derivations
Aggregate initialization nesting	32 levels
Local Initializers	150 levels (approximately)
Nesting of if statements, switch statements, and loops	32 levels

Notes: 1) This limit reflects the number of characters after splicing of \ lines. This limit also applies to any single macro definition or invocation.
2) This limit reflects the number of characters before concatenation. All other character strings are unrestricted.
3) This limit includes argument substitutions.
4) This limit includes –i and C_DIR directories.

Chapter 3

PP Runtime Environment

This section describes the TMS320C8x PP C runtime environment. To ensure successful execution of PP C programs, it is critical that all runtime code maintain this environment. If you write assembly language functions that interface to PP C code, follow the guidelines in this chapter.

Topics in this chapter include:

Topics

3.1 Memory Model

The PP C compiler treats memory as a single linear block that is partitioned into smaller blocks of code and data. Each block of code or data that a PP C program generates will be placed in its own contiguous space in memory. The PP compiler assumes that a full 32-bit address space is available in target memory.

Note: The Linker Defines the Memory Map

The linker, *not the PP compiler*, defines the memory map and allocates code and data into target memory. The PP compiler assumes nothing about the types of memory available, about any locations not available for code or data (holes), or about any locations reserved for I/O or control purposes. The PP compiler does assume that some variables will be located within a 15-bit scaled offset from address 0. The PP compiler produces relocatable code, thus allowing the linker to allocate code and data into the appropriate memory spaces. For example, you can instruct the linker to allocate global variables into on-chip RAM or to allocate executable code into external ROM. Each block of code or data could be allocated individually into memory, but this is not a general practice (an exception to this is memory-mapped I/O, although physical memory locations can be accessed with C pointer types).

3.1.1 PP Sections

The PP compiler produces several relocatable blocks of code and data. These blocks, called *sections*, can be allocated into memory in a variety of ways, to conform to a variety of system configurations. For more information about sections, please read Chapter 12, *Introduction to Common Object File Format*.

There are two basic types of sections:

- **Initialized** sections contain data or executable code. The PP C compiler creates four initialized sections: .ptext, .pcinit, .const, and .switch.
  - The **.ptext** section is an initialized section that contains all the executable code for the pp.
  - The **.pcinit** section is an initialized section that contains tables for initializing variables and constants.

- The **.const** section is an initialized section that contains string literals, and exists only if the –mc option has been used.
- The **.switch** section is an initialized section that contains switch tables, and exists only if the –mc option has been used.

❑ **Uninitialized** sections reserve space in memory (usually RAM). A program can use this space at runtime for creating and storing variables. The PP compiler creates four uninitialized sections: .pbss, .bss, .pstack, and .psysmem.

- The **.pbss** section reserves space for global and static variables. At program startup, the C boot routine copies data from the .pcinit section (which may be in ROM) and stores it in the .pbss section.
- The **.bss** section reserves space for global and static variables that have been defined as *far.* At program startup, the C boot routine copies far data from the .pcinit section and stores it in the .bss section.
- The **.pstack** section allocates memory for the system stack. This memory is used to pass arguments to functions and to allocate local variables.
- The **.psysmem** section is a memory pool, or heap, used by the dynamic memory functions: malloc, calloc, and realloc. If a C program does not use these functions, the PP compiler does not create the .psysmem section.

The .ptext, .pcinit, .const, and .switch initialized sections can be linked into any system memory. The .bss uninitialized section can be linked into any system RAM. The .pbss, .pstack, and .psysmem sections should be linked into the on-chip data or parameter RAM memory. You **can** link the .pstack and .psysmem sections into external memory, but you will notice a severe performance drop if you do.

For more information about allocating sections into memory, see Chapter 12, *Introduction to Common Object File Format*. and Chapter 13, *Linker Description*

3.1.2 C System Stack

The PP C compiler uses a stack to:

- Save function return addresses
- Allocate local variables
- Pass arguments to functions
- Save temporary results

The runtime stack grows down from high addresses to lower addresses. The PP compiler uses a register called the **SP** or **stack pointer** to manage this stack. The SP points to the current top of the stack.

The stack size is set by the linker. The linker also creates a global symbol, $_STACK_SIZE, and assigns it a value equal to the size of the stack in bytes. The default stack size is 128 bytes. You can change the size of the stack at link time by using the –pstack option on the linker command line and specifying the size of the stack as a constant immediately after the option. For more information on the –pstack option, see subsection 13.3.11, *Define PP Stack Size (–pstack* size *Option)*.

At system initialization, the SP is set to a designated address for the top-of-stack. This address is the first location past the end of the .pstack section. Since the position of the stack depends on where the .pstack section is allocated, the actual address of the stack is determined at link time.

The PP compiler uses the stack pointer (SP) to mark the top of the stack. The C environment automatically decrements SP at the entry to a function to reserve all the space necessary for the execution of that function. The stack pointer is then incremented at the exit of the function to restore the stack to its state before the function was entered. If you interface assembly language routines to C programs, be sure to restore the stack pointer to the same state it was in before the function was entered. (For more information about using the stack pointer, see Section 3.3, *PP Register Conventions*; for more information about the stack, see Section 3.4, *Function Structure and Calling Conventions*.)

Note: Stack Overflow

The PP compiler provides no means to check for stack overflow during compilation or at runtime. A stack overflow will disrupt the runtime environment, causing your program to fail. Be sure to allow enough space for the stack to grow.

3.1.3 Static and Global Memory Models

By default, the PP compiler reserves space for globals and statics in the .pbss section. The PP compiler assumes that the .pbss section is allocated in on-chip parameter or data RAM. This internal allocation allows the PP compiler to use the efficient PP relative addressing mode to load or store these variables.

For example, the following definition for the variable a_index:

```
static int a_index;
```

causes the PP compiler to generate the following directive:

```
$a_index: .usect .pbss, 4, 4
```

which reserves four bytes, aligned to a four-byte boundary, in the .pbss section. The PP compiler would then load the value of a_index using the following instruction:

```
d1 =w *(xba + $a_index)
```

This instruction uses the 15-bit scaled value of a_index and adds it to the base address of the current PP's parameter or data RAMs (depending on where the .pbss section was linked). The current PP is the PP on which the code is currently running. This addressing scheme is effective because it is efficient; however, the on-chip memory is limited and can be exhausted. If this is a problem for you, you can override the default behavior by using the *far* keyword in the object's declaration. The far keyword tells the PP compiler that the object may be located in external memory.

For example, the following definition of the variable index2:

```
far static int index2;
```

causes the PP compiler to generate the following directive:

```
   .bss $index2 4,4
```

which reserves four bytes, aligned to a four-byte boundary, in the .bss section. The PP compiler would then load the value of index2 using the following instructions:

```
a0 = $index2
nop
d1 =w *a0
```

Because of the far keyword, $index2 is a full 32-bit address and can be linked anywhere in the memory map.

The far keyword expands the memory available to the PP compiler for these variables, but accessing them will require two assembly language instructions, the use of the transfer controller, and, possibly, a nop, so references to variables defined in this way will cause substantial delays when compared to conventionally defined variables.

3.1.4 Dynamic Memory Allocation

The runtime-support library supplied with the PP compiler contains several functions (such as malloc, calloc, and realloc) that allow you to dynamically allocate memory for variables at runtime. Dynamic allocation is not a standard part of the C language; it is provided by standard runtime-support functions.

Memory is allocated from a global pool, or heap, that is defined in the .psysmem section. You can set the size of the .psysmem section by using the –pheap *size* option on the linker command line when you link your program. Specify the *size* as a constant. The linker also creates a global symbol, $_SYSMEM_SIZE, and assigns it a value equal to the size of the heap in bytes. The default size is 128 bytes. For more information on the –pheap option, see subsection 13.3.10, *Define PP Heap Size (–pheap* size *Option)*.

Dynamically allocated objects are not addressed directly (they are always accessed with pointers), and the memory pool is in a separate section; therefore, the dynamic memory pool can have a size limited only by the amount of memory in your system. To conserve space in the .pbss, you can allocate large arrays from the heap instead of declaring them as global or static. For example, instead of a declaration such as:

```
struct big table[100];
```

use a pointer and call the malloc function:

```
struct big *table
table = (struct big *)malloc(100*sizeof(struct big));
```

Note: .psysmem With PP Memory Allocation Functions

You should link the .psysmem section into on-chip data or parameter RAM, in order to execute the PP versions of memory allocation functions such as malloc, calloc, etc. You can link .psysmem into external memory if more memory space is required, but there will be a significant performance penalty for doing so.

The default alignment of a memory block returned by malloc or calloc is on a 4-byte boundary. If this alignment is not sufficient, use the memalign library function. See Section 5.5, *Runtime-Support Functions* for more information.

3.1.5 The ROM Model

The PP C compiler produces code that is suitable for use as firmware in a ROM-based system. In such a system, the initialization tables in the .pcinit section are stored in ROM. At system initialization time, the C boot routine copies data from these tables from ROM to the initialized variables in .pbss (RAM).

The MP compiler supports an additional model, called the RAM model. The PP compiler does not support this model.

3.2 Object Representation

This section explains how various data objects are sized, aligned, and accessed.

3.2.1 Data Type Storage

The following chart illustrates register and memory storage in the PP for various data types:

Table 3–1. PP Data Representation in Registers and Memory

Data Type	Register Storage	Memory Storage
char	bits 0–7 of register	1 byte
unsigned char	bits 0–7 of register	1 byte
short	bits 0–15 of register	2 bytes
unsigned short	bits 0–15 of register	2 bytes
int	bits 0–31 of register	4 bytes
unsigned int	bits 0–31 of register	4 bytes
enum	bits 0–31 of register	4 bytes
long	bits 0–31 of register	4 bytes
unsigned long	bits 0–31 of register	4 bytes
float	bits 0–31 of register	4 bytes
double (PP)	bits 0–31 of register	4 bytes
long double (PP)	bits 0–31 of register	4 bytes
struct	members stored as their individual types require	multiple of 4 bytes aligned to 32-bit boundary. Members stored as their individual types require
array	members stored as their individual types require	members stored as their individual types require

Figure 3–1. Data Representation in Registers for the TMS320C8x PP

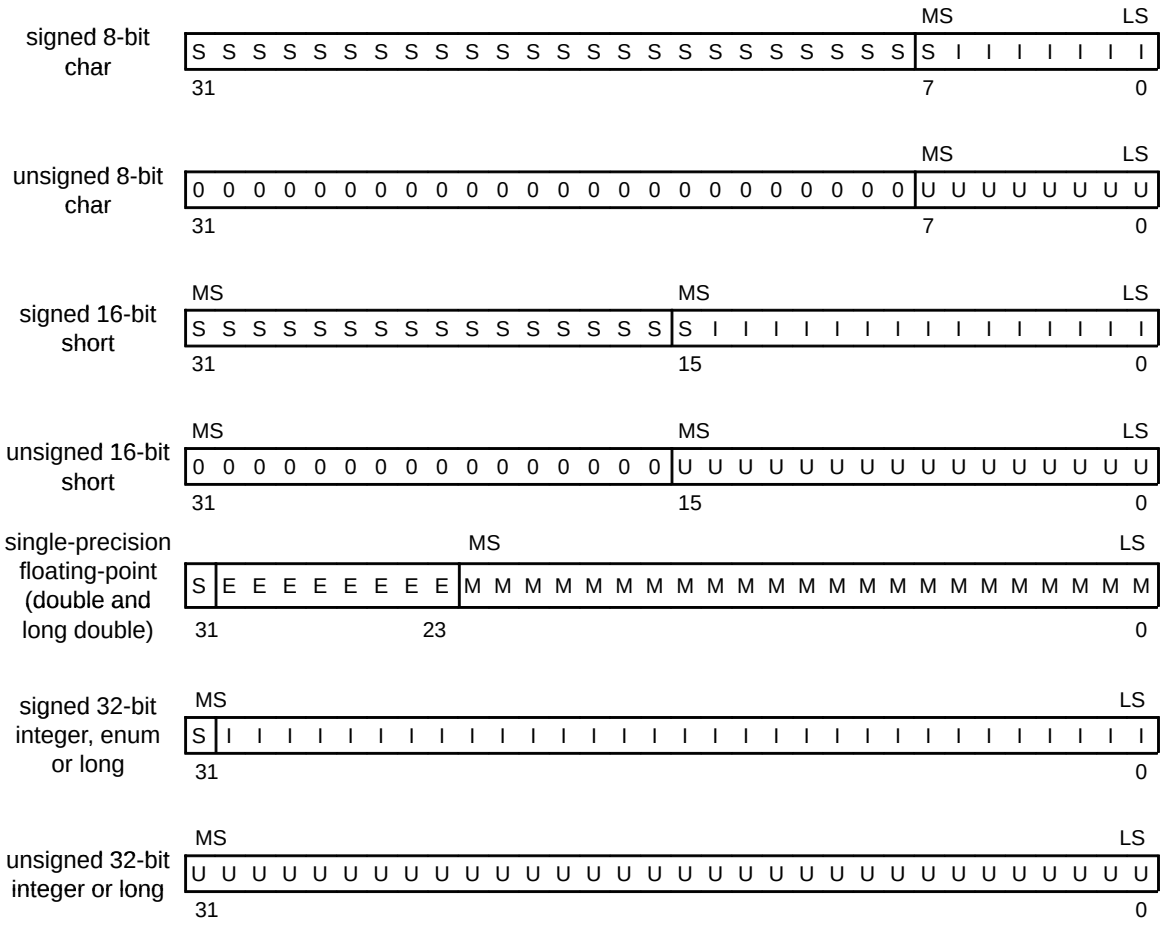


Figure 3–1 illustrates data storage shown in Table 3–1. In big-endian mode, data objects are loaded to registers by moving the first byte (that is, lowest address) of memory to the highest eight bits reserved in the register, with subsequent bytes occupying lower byte values. In little-endian mode, the first byte is moved to the lowest eight bits of the register with subsequent bytes occupying higher byte values.

Structures will be aligned on a 32-bit boundary, and will always reserve a multiple of four bytes of storage in memory. That is, even a structure composed only of one char will reserve four bytes of storage. Members of structures are stored in the same manner as if they were individual objects.

Arrays will be aligned on a boundary required by their element types. Elements of arrays are stored in the same manner as if they were individual objects.

3.2.2 PP Bit Fields

Bit fields are the only objects that are packed; that is, two bit fields may be stored in the same byte. Bit fields may range in size from 1 to 32 bits, but they will never span a 4-byte boundary.

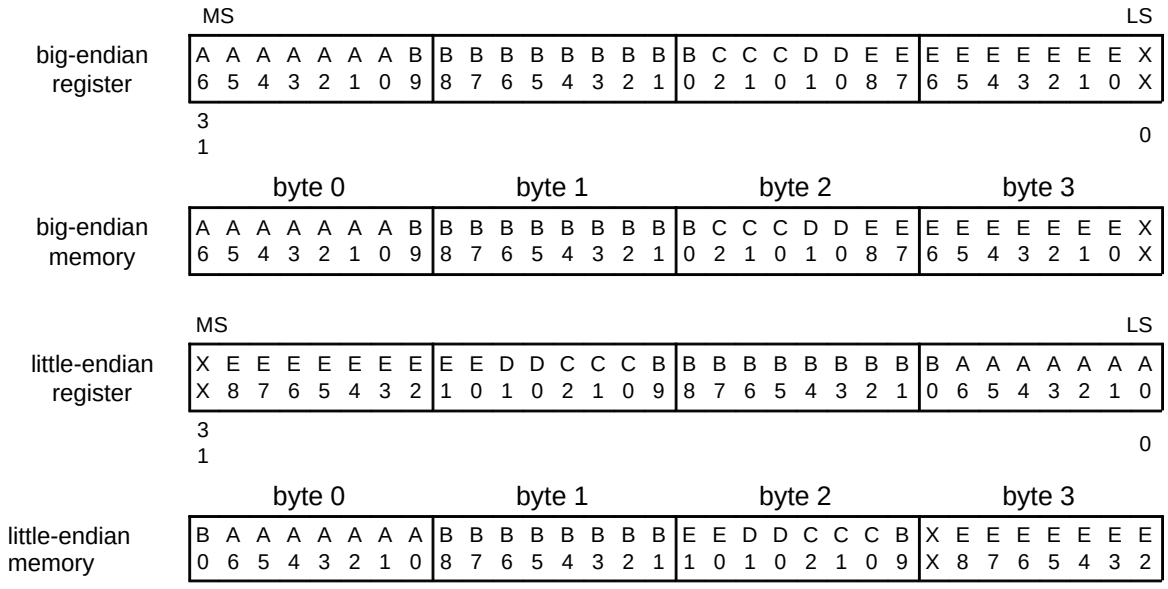
For big-endian mode, bit fields are packed into registers from MSB to LSB in the order in which they are declared, and packed in memory from MSbyte to LSbyte. (see Figure 3–2). For little-endian mode, bit fields are packed in registers from the LSB to the MSB in the order in which they are declared, and packed in memory from LSbyte to MSbyte (see Figure 3–2).

In the following example of bit field packing, assume these bit field declarations:

```
struct{
   int A:7
   int B:10
   int C:3
   int D:2
   int E:9
}x;
```

Note that A0 represents the least significant bit of the field A; A1 represents the next least significant bit, etc. Note again that memory storage for bit fields is done with a byte-by-byte, rather than bit-by-bit, transfer.

Figure 3–2. Bit Field Packing in Big-Endian and Little-Endian Formats



3.2.3 Character String Constants

In C, a character string constant can be used in one of two ways:

- It can initialize an array of characters; for example:

```
char s[] = ”abc”;
```

- It can be used in an expression; for example:

```
strcpy (s, ”abc”);
```

When a string is used as an initializer, it is simply treated as an initialized array; each character is a separate initializer. For more information about initialization, see Section 3.8, *System Initialization*.

When a string is used in an expression, the string itself is defined in one of two ways, depending on the use of the –mc PP compiler option.

If –mc is not used, then space is reserved for strings in on-chip memory in the .pbss section, and they are autoinitialized as though they were an array defined in the program. For example, the following lines define the string xyz, along with the terminating byte; the label SL4 points to the string:

```
SL4:   .usect    .pbss 4, 1
       .sect     ”.pcinit”
       .align    8
       .field    4, 32
       .field    SL4+0, 32
       .align    1
       .string   ”xyz”, 0
```

For more information about autoinitialization, see subsection 3.8.1, *PP Autoinitialization of Variables and Constants.*

If –mc is used, then strings are defined in the .const section using the .string assembler directive, and the terminating 0 byte is included. A unique label points to the string. For example, the following lines define the string abc, along with terminating byte: (The label SL5 points to the string.)

```
       .sect ”.const”
SL5:   .string ”abc”,0
```

The .const section will be allocated in external memory, so functions compiled with –mc will reference strings very slowly. The benefit of –mc is that it allows you to reserve the internal data and parameter RAMs for more important data. For more information about options, see subsection 1.1.3, *Compiler Options*.

String labels have the form **SL***n*, where *n* is a number assigned by the PP compiler to make the label unique. The number begins at 0 and is increased by 1 for each string defined. All strings used in a source module are defined at the end of the compiled assembly language module.

The label SL*n* represents the address of the string constant. The PP compiler uses this label to reference the string expression.

If the same string is used more than once within a source module, the string will not be duplicated in memory. All uses of an identical string constant share a single definition of the string.

Because strings may be stored in the .const section (possibly in ROM) and shared, it is bad practice for a program to modify a string constant. The following code is an example of incorrect string use:

```
char  *a = ″abc″;
a[1] = ’x’;        /* Incorrect!   */
```

3.3 PP Register Conventions

The compilers follow strict conventions that associate specific registers with specific operations in the C environment. If you plan to interface an assembly language routine to a C program, it is important that you understand these register conventions.

The register conventions dictate both how the PP compiler uses registers and how values are preserved across function calls. When a function call occurs, the called function is responsible for preserving the contents of certain registers. The caller need not save the contents of those registers. If the caller is using a register that is not preserved by the called function, the caller must save the register's value before making the call.

The following table summarizes how the PP compiler uses the TMS320C8x PP registers and shows which registers are defined to be preserved across function calls.

Table 3–2. Registers That May Be Used or Modified by Compiled Code

Register	Preserved by Called Function	Special Uses	
d0	No		
d1	No	First argument	
d2	No	Second argument	
d3	No	Third argument	
d4	No	Fourth argument	
d5	No	Int and float return	
ls0–ls2	No	Loop start	
le0–le2	No	Loop end	
lc0–lc2	No	Loop count	
lr0	No	Implicit use with lrs0	
lr1	No	Implicit use with lrs1	
lr2	No	Implicit use with lrs2	
sp	Yes	Stack pointer	
zero	—	Hardwired zero	
sr	No	Status register	
iprs	No	Return address register	
lctl	No	Loop control – implicit with lrsx	
a8	No	Pointer return	
Register	**Preserved By Called Function**	**Register**	**Preserved By Called Function**
d6	Yes	d7	Yes
a0–a3	No	a4	Yes
a9–a11	No	a12	Yes
x0–x2	No	x8–x10	No

3.3.1 Register Variables and Register Allocation

Register variables are local variables or PP compiler temporaries defined to reside in a register rather than in memory. Storing local variables in registers allows significantly faster access, which improves the efficiency of compiled code.

The registers in Table 3–2 are available to the PP compiler for register variables and also for temporary expression results. If the PP compiler cannot allocate a register of a required type, then spilling will occur. Spilling is the process of moving a register's contents to memory to free the register for another purpose.

With a few exceptions, registers are not preserved across calls. Whenever a called function uses registers d6, d7, a4, or a12, it must save the contents of these registers on entrance, then restore the contents on exit. This ensures that a called function does not disrupt the register variables of the caller.

When you are not using the optimizer (–o option), you can allocate register variables with the register keyword.

When you are using the optimizer, the PP compiler ignores the register keyword and uses a cost analysis algorithm to allocate variables and temporaries to registers.

3.3.2 Control Registers

You can use the cregister keyword to access the PP control registers from C. If you use one of these registers in C code and the register may be changed by an external mechanism, then you should declare the register to be volatile. For example, to read the INTFLG register:

```
cregister extern volatile unsigned int INTFLG;

main()
{
   unsigned int x = INTFLG;
}
```

The cregister keyword is only allowed on declarations that occur outside of a function. It is also only allowed on a variable declaration with an integer type.

The PP control registers are COMM, INTFLG, INTEN, MF, TAG0, TAG1, TAG2, and TAG3. They are declared in the header file mvp.h which is shipped with the compiler.

For more information, see Section 2.5, *The cregister Keyword*.

3.4 Function Structure and Calling Conventions

The PP C compiler imposes a strict set of rules on function calls. Any function that calls or is called by a C function must follow these rules. Failure to adhere to these rules can disrupt the C environment and cause a program to fail.

3.4.1 Responsibilities of the Calling Function

A function performs the following tasks when it calls another function.

1) If any arguments are to be passed to a function, then the first four arguments, or as many as exist, are placed in registers d1, d2, d3, and d4 in order. Any remaining arguments are placed on the stack in reverse order (the stack pointer points to the 5th argument and SP + <offset> points to the 6th argument, and so on.) Arguments pushed on the stack must be aligned to a value appropriate for their size. Any argument that is not defined in a prototype and whose size is less than 32 bits is passed as a 32-bit value.

   A structure argument is passed as the address of the structure. It is up to the called function to make a local copy.

   If a function is declared with an ellipse, indicating that it can be called with varying numbers of arguments, the convention is slightly modified. The last explicitly declared argument is passed on the stack, so that its stack address can act as a reference for accessing the undeclared arguments. For example:

   ```
   int vararg(int a, int b, int c, ...)
   ```

   causes a to be placed in register d1, b in register d2, and c on the stack at the location pointed to by the stack pointer; any other arguments are also placed on the stack.

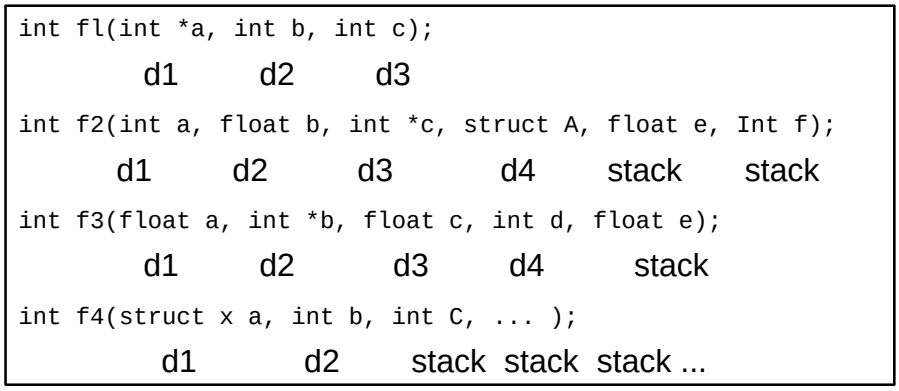
   The called function is responsible for preserving d6, d7, a4, a12, and SP. Any other registers whose contents must be preserved must be pushed onto the stack.

2) The caller calls the function.

3) When the called function is complete, the caller restores the stack pointer to its value at the function call by adding to the stack pointer. This does not occur in assembly compiled from C code, because the stack space needed for all calls is allocated at the beginning of the function and deallocated at the end of the function.

Figure 3–3 shows some sample function declarations. Under each function declaration is a notation showing where each parameter is passed.

Figure 3–3. Register Argument Conventions



3.4.2 Responsibilities of a Called Function

A called function must perform the following tasks:

1) The called C function must allocate enough space on the stack for any local variables, temporary storage areas, and arguments to functions that this function may call. For assembly compiled from C code, this allocation occurs once at the beginning of the function by subtracting a constant from the SP register. The SP register will not be decremented anywhere else within this function.

2) If the called function calls any other functions, then the return address must be saved on the stack. Otherwise, it is left in the IPRS register and it will be overwritten by the next function call.

3) If the called function modifies any save-on-entry registers (d6, d7, a4, or a12), it must save them, either in other registers or on the stack. The called function may modify any other registers without saving them.

4) If the called function is expecting a structure argument, it will be passed a pointer to the structure instead. If any writes are made to the structure from within the called function, then space for a local copy of the structure must be allocated on the stack, and the local structure must be copied from the passed pointer to the structure. If no writes are made to the structure, then it can be referenced in the called function indirectly through the pointer argument.

   You must be careful to properly declare functions that accept structure arguments both at the point where they are called

(so that the structure argument is passed as an address) and at the point where they are defined (so the function knows to copy the structure to a local copy.)

5) The called function executes its code.

6) If the called function returns an integer or a floating-point value, the return value will be placed in the d5 register. If the function returns a pointer, the value will be placed in the a8 register.

   If the function returns a structure, the caller allocates space for the structure and then passes the address to the function as the first argument. To return a structure, the called function copies the structure to the memory block pointed to by the extra argument.

   In this way, the caller can be smart about telling the called function where to return the structure. For example, in the statement:

   ```
   s = f(x)
   ```

   where s is a structure and f is a function that returns a structure, the caller can actually write the call as:

   ```
   f(&s, x)
   ```

   The function f then copies the return structure directly into s, performing the assignment automatically.

   If the caller does not use the return structure value, then an address value of 0 can be passed as the first argument, This directs the called function not to copy the return structure.

   You must be careful to properly declare functions that return structures both at the point where they are called (so that the extra argument is passed) and at the point where they are defined (so the function knows to copy the result.)

7) Any save-on-entry register that is saved in step 3 is restored.

8) The space allocated for the function in step 1 is reclaimed at the end of the function by adding the same constant subtracted in step 1) to the SP register.

9) The function returns by writing the value of IPRS or the saved value of IPRS to the PC.

3.4.3 Accessing Arguments and Local Variables

A function accesses its stack arguments and local nonregister variables indirectly through the stack pointer (SP), which always points to the top of the stack. Since the stack grows toward smaller addresses, the local and argument data for a function are accessed with a positive offset from the SP register. Local variables, temporary storage, and the area reserved for stack arguments to functions are accessed with offsets smaller than the constant subtracted from the SP at the beginning of the function. Stack arguments passed to this function are accessed with offsets greater than or equal to the constant subtracted from the SP at the beginning of the function.

The PP compiler attempts to keep register arguments in their original registers if the optimizer is used or if they are declared with the register keyword. Otherwise the arguments are copied to the stack in order to free those registers for further allocation.

3.5 Interfacing PP C With Assembly Language

There are two ways to use assembly language with C code:

- Use separate modules of assembled code and link them with compiled C modules (subsection 3.5.1, following). This is the most versatile method.
- Use inline assembly language, embedded directly in the C source (subsection 3.5.3, *Inline Assembly Language*).

3.5.1 Interfacing PP C With Assembly Language Modules

Interfacing with assembly language functions is straightforward if you follow the calling conventions defined in Section 3.4, *Function Structure and Calling Conventions*, and the register conventions defined in Section 3.3, *PP Register Conventions*. C code can access variables and call functions defined in assembly language, and assembly code can access C variables and call C functions. Follow these guidelines to interface assembly language and C:

- All functions, whether they are written in C or assembly language, must follow the conventions outlined in Section 3.3, *PP Register Conventions*.
- You must preserve any dedicated registers that will be modified by the called function. The dedicated registers are d6, d7, a4, a12, and SP. If the SP register is used normally, it does not need to be explicitly preserved. In other words, the assembly function is free to use the stack as long as anything that is pushed is popped off again before the function returns, thus preserving SP.
- When calling a C function from assembly language, load the first four arguments into d1, d2, d3, and d4, and push the remaining arguments on the stack as described in subsection 3.4.2, *Responsibilities of a Called Function*. When calling C functions, remember that only the dedicated registers (d6, d7, a4, a12, and SP) are preserved. C functions can change the contents of any other register. Any register whose contents should be preserved must be pushed on the stack before the function call, and restored after the function returns.
- Interrupt routines must save the registers they use. For more information about interrupt handling, refer to Section 3.6, *Interrupt Handling*.

- Functions must return values correctly according to their C declarations. Integers and floating-point values are returned in d5. Pointers are returned in A8, and structures are returned as described in subsection 3.4.2, *Responsibilities of a Called Function*.

- No assembly module should use the .pcinit section for any purpose other than autoinitialization of global variables. The C startup routine in boot.c assumes that the .pcinit section consists entirely of initialization tables. Disrupting the tables by putting other information in .pcinit can cause unpredictable results.

- The PP compiler appends a dollar sign ($) to the beginning of all identifiers. In assembly language modules, you must use a prefix of $ for all objects that are to be accessible from C. For example, a C object named x is called $x in PP assembly language. For identifiers that are to be used only in an assembly language module or modules, any name that does not begin with a leading dollar sign may be safely used without conflicting with a C identifier.

- Any object or function declared in assembly that is to be accessed or called from C must be declared with the .global or .sysmem directive in the assembler. This defines the symbol as external and allows the linker to resolve references to it.

  Likewise, to access a C function or object from assembly, declare the C object with .global or .system. This creates an undefined external reference that the linker will resolve.

An example of an assembly language function

Example 3–1 illustrates a C function called main, which calls an assembly language function called asmfunc. The asmfunc function takes its single argument, adds it to the C global variable called gvar, and returns the result.

Example 3–1. An Assembly Language Function

C program

```
extern int asmfunc(int i); /* declare external function */
int gvar;                  /* define global variable    */

main()
{
   int i;
   i = 100;
   i = asmfunc(i);   /* call function normally        */
}
```

Assembly Language program (PP)

```
   .global $asmfunc       ; Declare external function
   .global $gvar          ; Declare external variable

$asmfunc:

   br = iprs
   d3 =w *(xba + $gvar)   ; load global variable
   d5 = d1 + d3           ; d1 is argument
                          ; d5 is return value
                          ; d3 is value of gvar
```

All C functions, including assembly language functions called by C, should have prototypes so that the arguments are passed correctly.

3.5.2 Accessing Assembly Language Variables From C

It is sometimes useful for a C program to access variables or constants defined in assembly language. There are three methods that you can use to accomplish this, depending on where and how the item is defined: a variable defined in a section linked to on-chip RAM, a variable defined in a section linked to external RAM, or a constant.

Accessing variables defined in a section linked to on-chip RAM

To access a variable from the .pbss section or some other section named with the .usect directive, which is linked to on-chip RAM:

- ❑ Use the .usect directive to define the variable.
- ❑ Use the .global directive to make the definition external.
- ❑ Remember to precede the name with a dollar sign in assembly language references.
- ❑ In C, declare the variable as *extern*, and access it normally.

Example 3–2 shows an example that accesses a variable that will be linked to on-chip RAM.

Example 3–2. Accessing a Variable Linked Into On-chip RAM

Assembly Language program (PP)

```
* Note the use of dollar signs in the
*         following lines

$var:  .usect ".pbss",4,4   ; Define the variable
       .global   $var       ; Declare it as external
```

C program

```
extern int var;     /* External variable       */
var = 1;            /* Use the variable        */
```

Accessing variables defined in a section linked to external RAM

If a variable is not defined in on-chip RAM, it is more difficult to access it from C. A common situation is a lookup table, defined in assembly, that you don't want to put in on-chip RAM. In this case, you must define a section for the table and access it from C as *far*.

To access a variable from a section that is linked to external RAM:

- Use the .sect or .usect directive to define the variable in assembly.
- Use the .global directive to make the definition external.
- Remember to precede the name with a dollar sign in assembly language references.
- In C, declare the variable as extern far and access it normally.

Example 3–3 shows a program for accessing a variable that will be linked into external RAM.

Example 3–3. Accessing a Variable Not Linked Into On-chip RAM

Assembly Language program (PP)

```
   .sect  "my_table" ; Create a section linked to
                     ; external RAM
   .global $table    ; Make table external
table:            ; Table starts here
       .word 5
       .word 10
       .word 9
```

C program

```
extern far int table[3];   /* This is the object */
int ref = table[1]; /* Declare entry for table */
```

Accessing assembly language constants

You can define global constants in assembly language by using the .set, .global, and .system directives or you can define them in a linker command file using a linker assignment statement. These constants are accessible from C only with the use of special operators.

For normal variables defined in C or assembly language, the symbol table contains the *address of the value* of the variable. For assembler constants, however, the symbol table contains the *value* of the constant. The PP compiler cannot tell which items in the symbol table are values and which are addresses.

If you try to access an assembler (or linker) constant by name, the PP compiler will attempt to fetch a value from the address represented in the symbol table. To prevent this unwanted fetch, you must use the & (address of) operator to get the value. In other words, if x is an assembly language constant, its value in C is &x.

You can use casts and #defines to ease the use of these symbols in your program, as in Example 3–4.

Example 3–4. Accessing an Assembly Language Constant From C

Assembly Language program (PP)

```
$table_size   .set   10000         ; define the constant
              .global $table_size ; make it global
```

C program

```
extern far int table_size;    /*external ref */

#define TABLE_SIZE ((int) (&table_size))

       .          /* use cast to hide address-of */
       .
       .

for (i=0; i<TABLE_SIZE; ++i)
                     /* use like normal symbol */
```

Since you are referencing only the symbol's value as stored in the symbol table, the symbol's declared type is unimportant. In Example 3–4, int is used. You can reference linker-defined symbols in a similar manner.

3.5.3 Inline Assembly Language

Within a C program, you can use the *asm statement* to inject a single line of assembly language into the assembly language file that the PP compiler creates. A series of asm statements places sequential lines of assembly language into the PP compiler output with no intervening code.

For more information on the asm statement, see Section 2.9, *The asm Statement*.

Note: Using the PP asm Statement

The asm statement is provided so you can access features of the hardware that would be otherwise inaccessible from C. When you use the asm statement, be extremely careful not to disrupt the C environment. The compiler does not check or analyze the inserted instructions.

Do not use the asm statement to insert assembler directives that would change the assembly environment.

The asm statement is also useful for inserting comments in the compiler output; simply start the assembly code string with a semicolon (;) as shown below:

```
asm(";*** this is an assembly language comment");
```

3.6 Interrupt Handling

As long as you follow the guidelines in this section, PP C code can be interrupted temporarily without disrupting the C environment. When the C environment is initialized, the startup routine does not enable or disable interrupts. If your system uses interrupts, it is your responsibility to handle any required enabling or masking of interrupts. Such operations have no affect on the C environment and can be easily incorporated with *asm* statements.

3.6.1 Saving Registers During Interrupts

When C code is interrupted, the interrupt routine must preserve the contents of all machine registers that are used by the routine or by any function called by the routine. This includes all the data registers, all the address registers, all the index registers, all the loop registers, and LCTL, SP, SR, and IPRS.

The compiler will handle register preservation if the interrupt service routine is written in C.

3.6.2 Using PP C Interrupt Routines

Interrupts can be handled directly with C functions by using the interrupt keyword on the function definition. Interrupt functions must be defined as returning void type, and may not have any arguments. For example, to define an interrupt function called timer:

```
interrupt void timer ()
{
...
}
```

The special name *c_int00* is the C entry point; this name is reserved as a special interrupt function name. c_int00 initializes the system and calls the function main. Because it has no caller, c_int00 does not save any registers.

If a C interrupt routine does not call any other functions, only those registers that are actually used in the interrupt handler are saved and restored. However, if a C interrupt routine *does* call other functions, these functions may modify unknown registers that are not used in the interrupt handler itself. For this reason, the routine may save *all* the expression registers if any other functions are called.

Interrupt handling functions should not be called directly. For more information, see Section 2.6, *The interrupt and trap Keywords*.

3.6.3 Assembly Language Interrupt Routines

Interrupts can also be handled with assembly language code, as long as the register conventions are followed. Like all assembly functions, interrupt routines can use the stack, access global C variables, and call C functions normally. When calling C functions, be sure that any register not saved across function calls listed in Table 3–2, is saved, because the C function can modify them. The PP compiler will preserve the values in d6, d7, a4, a12, and SP, so these registers need not be saved explicitly in assembly.

3.7 Runtime-Support Arithmetic Routines

The runtime-support library contains a number of assembly language functions that provide arithmetic routines for C math operations that the PP instruction set does not provide, such as integer division, integer modulus, and floating-point operations.

These routines follow the standard C calling sequence and could be called directly from C, except that the function names do not begin with a dollar sign in the assembly language files they are defined in.

The source code for these functions is in the source library pp_rts.src. The source code has comments that describe the operation of the functions. You can extract, inspect, and modify any of the math functions; be sure you follow the calling conventions and register-saving rules outlined in this chapter.

Table 3–3 summarizes the runtime-support functions used for arithmetic.

Table 3–3. Summary of Runtime-Support Arithmetic Functions

Function		Description	Defined In
int	F_FTOI (float)	Float to signed int conversion	f_ftoi.asm
uint†	F_FTOU (float)	Float to unsigned int conversion	f_ftou.asm
float	F_ITOF (int)	Signed int to float conversion	f_itof.asm
float	F_UTOF (uint)	Unsigned int to float conversion	f_utof.asm
float	F_ADD (float, float)	Floating-point addition	f_sub.asm
float	F_SUB (float, float)	Floating-point subtraction	f_sub.asm
float	F_MPY (float, float)	Floating-point multiplication	f_mpy.asm
float	F_DIV (float, float)	Floating-point division	f_div.asm
float	F_CMP (float, float)	Floating-point comparison	f_cmp.asm
int	I_DIV (int, int)	Signed integer division	i_div.asm
int	I_MOD (int, int)	Signed integer modulus	i_mod.asm
uint	U_DIV (uint, uint)	Unsigned integer division	u_div.asm
uint	U_MOD (uint, uint)	Unsigned integer modulus	u_mod.asm

† Note: uint is defined as typedef unsigned int uint.

The source files for these functions are archived in pp_rts.src.

3.8 System Initialization

Before you can run a C program, the C runtime environment must be created. This task is performed by the C boot routine, which is a function called c_int00.

The c_int00 function can be branched to, called, or vectored by reset hardware to begin running the system. The function is in the runtime-support library and must be linked with the other C object modules. This occurs automatically when you use the –pc option in the linker and include a runtime-support library created from pp_rts.src (that is, pp_rts.lib, pprtsl.lib, or a library you create) as one of the linker input files. When C programs are linked, the linker sets the entry point value in the executable output module to the symbol c_int00.

The c_int00 function performs the following tasks to initialize the environment:

1) It defines a section called .pstack for the system stack and sets up the initial stack pointer.

2) It autoinitializes global variables by copying the data from the initialization tables in .pcinit to the storage allocated for the variables in .bss or .pbss.

3) It calls the function main to begin running the C program.

You can replace or modify the boot routine to meet your system requirements. However, the boot routine *must* perform the three operations listed above to correctly initialize the C environment. The runtime-support source library contains the source to this routine in a module named boot.asm.

3.8.1 PP Autoinitialization of Variables and Constants

Some global variables must have initial values assigned to them before a C program starts running. The process of retrieving these variables' data and initializing the variables with the data is called autoinitialization.

The PP compiler builds tables in a special section called .pcinit that contains data for initializing global and static variables. Each compiled module contains these initialization tables. The linker combines them into a single table (a single .pcinit section). The boot routine uses this table to initialize all the variables that need values before the program starts running.

Note: Initializing Variables

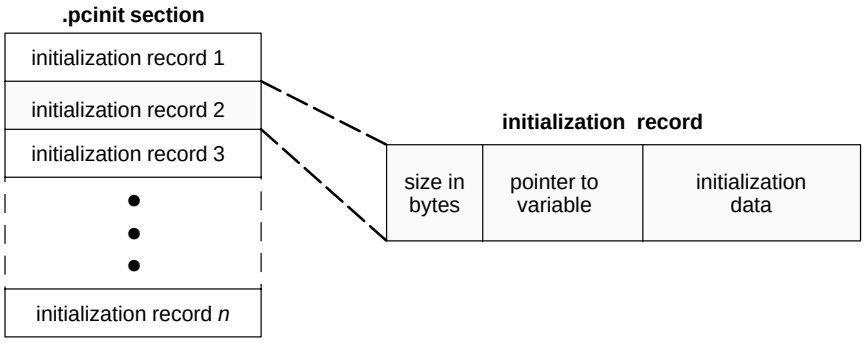
In standard C, global and static variables that are not explicitly initialized are set to 0 before program execution. The PP C compiler does not perform any preinitialization of uninitialized variables. Any variable that must have an initial value of 0 must be explicitly initialized.

For more information on copying the initialization data into memory, see *ROM initialization model for the PP,* page CG:3-31.

Initialization tables

The tables in the .pcinit section consist of initialization records with varying sizes. Each initialization record has the following format:

Figure 3–4. Format of Initialization Records in the .pcinit Section



- The first field of an initialization record is the size (in bytes) of the initialization data. If the size is negative, then the data is PP address patch data (described below).
- The second field is the starting address of the area where the initialization data must be copied. (This field points to a variable.)
- These first two fields are followed by one or more bytes of data. During autoinitialization, this data is copied to the specified address of a variable.

Each variable that must be autoinitialized has an initialization record in the .pcinit section.

If the first field is negative, then the record represents a list of addresses that need to be patched with the appropriate PP number. This is required since all variables that are linked into on-chip RAM are assumed to be relative to address 0, and the

appropriate relocation is done at runtime through the use of the PP relative addressing mode *(xba + sym). Since the PP relative addressing mode is not used for autoinitialization, this relocation is achieved with the PP address patch autoinitialization record. The PP address patch autoinitialization record has the following fields:

- A negative size in bytes of the list of addresses
- A list of addresses to be patched

Each variable that is autoinitialized with the address of an on-chip variable will be in the PP address patch list.

Example 3–5 shows initialized variables defined in C.

Example 3–5. PP Initialization Table Address Patch List

```
int x;
int i = 23;
int *p = &x;
int a[5] = {1,2,3,4,5}
```

The initialization information for these variables is:

```
   .sect  ".pcinit"
   .align 8          ; each record aligned on 8-byte boundary
   .field 4, 32      ; length of data is 4 bytes
   .field $i+0,32    ; address of i
   .field 23, 32     ; data is 23

   .sect  ".pcinit"
   .align 8
   .field 4, 32      ; length of data is 4 bytes
   .field $p+0,32    ; initialize p
   .field x, 32      ; dat is value of x

   .sect  ".pcinit"
   .align 8
   .field IR1,32     ; length is 20
   .field $a+0,32    ; address is a
   .field 1,32       ; $a[0] data is 1
   .field 2,32       ; $a[1] data is 2
   .field 3,32       ; $a[2] data is 3
   .field 4,32       ; $a[3] data is 4
   .field 5,32       ; $a[4] data is 5
IR1: .set    20

   .sect  ".pcinit"
   .align 8
   .field –4,32      ; size of –4 indicates 1 4-byte patch
                       address
   .field $p,32      ; patch 4-byte value at address p
                     ;  with pp #
```

The .pcinit section contains *only* initialization tables in this format. When interfacing assembly language modules, do not use the .pcinit section for any other use.

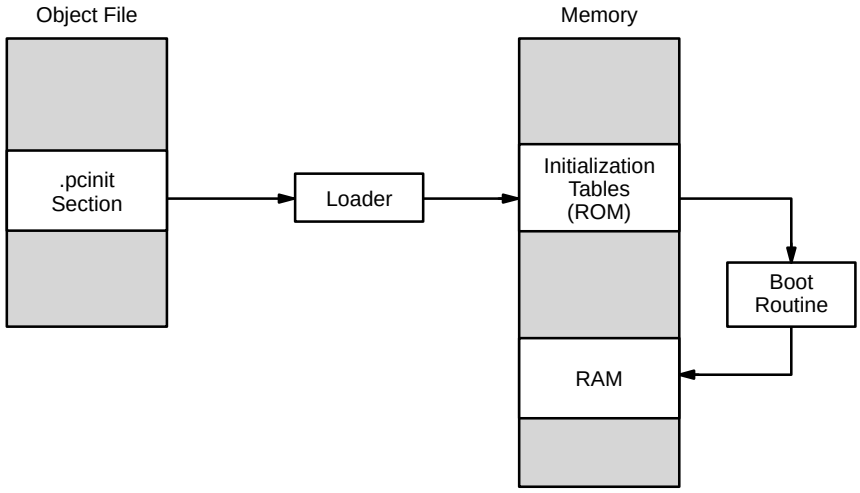
When you link a program with the –pc option, the linker links together the .pcinit sections from all the C modules and appends a null word to the end of the entire section. This appears as a record with a size field of 0 and marks the end of the initialization tables.

ROM initialization model for the PP

The ROM model is the PPs only model for autoinitialization. To use the ROM model on the PP, invoke the linker with the –pc option.

The .pcinit section is loaded into memory (possibly ROM) along with all the other sections, and global variables are initialized at runtime. The linker defines a special symbol called $pcinit that points to the beginning of the tables in memory. When the program begins running, the C boot routine copies data from the tables (pointed to by $pcinit) into the specified variables. This allows initialization data to be stored in ROM and then copied to RAM each time the program is started.

Figure 3–5. ROM Model of Autoinitialization



Chapter 4

MP Runtime Environment

This chapter describes the TMS320C8x MP C runtime environment. To ensure successful execution of MP C programs, it is critical that all runtime code maintain this environment. If you write assembly language functions that interface to C code, follow the guidelines in this chapter.

Topics

4.1 Memory Model

The MP C compiler treats memory as a single linear block that is partitioned into subblocks of code and data. Each block of code or data that a C program generates will be placed in its own contiguous space in memory. The MP compiler assumes that a full 32-bit address space is available in target memory.

Note: The Linker Defines the Memory Map

The **linker**, *not the MP compiler*, defines the memory map and allocates code and data into target memory. The MP compiler assumes nothing about the types of memory available, about any locations not available for code or data (holes), or about any locations reserved for I/O or control purposes. The MP compiler produces relocatable code that allows the linker to allocate code and data into the appropriate memory spaces. For example, you can use the linker to allocate global variables into on-chip RAM or to allocate executable code into external ROM. Each block of code or data could be allocated individually into memory, but this is not a general practice (an exception to this is memory-mapped I/O, although physical memory locations can be accessed with C pointer types).

4.1.1 MP Sections

The MP compiler produces seven relocatable blocks of code and data; these blocks, called *sections*, can be allocated into memory in a variety of ways, to conform to a variety of system configurations. For more information about sections, see Chapter 12, *Introduction to Common Object File Format*.

There are two basic types of sections:

- **Initialized sections** contain data or executable code. The MP C compiler creates four initialized sections: .text, .cinit, .const, and .switch.
  - The **.text section** is an initialized section that contains all the executable code for the MP.
  - The **.cinit section** is an initialized section that contains tables for initializing variables and constants.
  - The **.const section** is an initialized section that contains string literals and floating-point constants.
  - The **.switch section** is an initialized section that contains switch tables.

- **Uninitialized sections** reserve space in memory (usually RAM). A program can use this space at runtime for creating and storing variables. The MP compiler creates three uninitialized sections, .bss, .stack, and .sysmem.
  - The **.bss section** reserves space for global and static variables. At program startup, the C boot routine copies data out of the .cinit section (which may be in ROM) and stores it in the .bss section.
  - The **.stack section** allocates memory for the system stack. This memory is used to pass arguments to functions and to allocate local variables.
  - The **.sysmem section** is a memory pool or heap used by the dynamic memory functions malloc, calloc, and realloc. If a C program does not use these functions, the MP compiler does not create the .sysmem section.

Note that the *assembler* creates three default sections (.text, .bss, and .data); the MP C compiler, however, does not use the .data section.

For more information about allocating sections into memory, see Chapter 12, *Introduction to Common Object File Format*.

4.1.2 C System Stack

The MP C compiler uses a stack to:

- Save function return addresses
- Allocate local variables
- Pass arguments to functions
- Save temporary results

The runtime stack grows down from high addresses to lower addresses. The MP compiler uses the R1 register to manage this stack. R1 is the **stack pointer** (**SP**); it points to the current top of the stack.

The stack size is set by the linker. The linker also creates a global symbol, __STACK_SIZE, and assigns it a value equal to the size of the stack in bytes. The default stack size is 1024 bytes. You can change the size of the stack at link time by using the –stack option on the linker command line and specifying the size of the stack as a constant immediately after the option. For more information on the –stack option, see subsection 13.3.14, *Define MP Stack Size (–stack* size *Option)*.

At system initialization, the SP is set to a designated address for the top-of-stack. This address is the first location past the end of the .stack section. Since the position of the stack depends on where the .stack section is allocated, the actual address of the stack is determined at link time.

The MP compiler uses the stack pointer (SP) to mark the top of the stack. The C environment automatically decrements SP at the entry to a function to reserve all the space necessary for the execution of that function. The stack pointer is then incremented at the exit of the function to restore the stack to its state before the function was entered. If you interface assembly language routines to C programs, be sure to restore the stack pointer to the same state it was in before the function was entered. (For more information about using the stack pointer, see Section 4.4, *MP Register Conventions*; for more information about the stack, see Section 4.5, *Function Structure and Calling Conventions*.)

Note: Stack Overflow

The MP compiler provides no means to check for stack overflow during compilation or at runtime. A stack overflow will disrupt the runtime environment, causing your program to fail. Be sure to allow enough space for the stack to grow.

4.1.3 Dynamic Memory Allocation

The runtime-support library supplied with the MP compiler contains several functions (such as malloc, calloc, and realloc) that allow you to dynamically allocate memory for variables at runtime. Dynamic allocation is not a standard part of the C language; it is provided by standard runtime-support functions.

Memory is allocated from a global pool or heap that is defined in the .sysmem section. You can set the size of the .sysmem section by using the –heap *size* option on the linker command line when you link your program. Specify the *size* as a constant. The linker also creates a global symbol, __SYSMEM_SIZE, and assigns it a value equal to the size of the heap in bytes. The default size is 1024 bytes. For more information on the –heap option, see subsection 13.3.6, *Define MP Heap Size (–heap* size *Option)*.

Dynamically allocated objects are not addressed directly (they are always accessed with pointers), and the memory pool is in a separate section; therefore, the dynamic memory pool can have a size limited only by the amount of available memory in your system. To conserve space in the .bss section, you can allocate large arrays from the heap instead of declaring them as global or static. For example, instead of a declaration such as:

```
struct big table[100];
```

use a pointer and call the malloc function:

```
struct big *table
table = (struct big *)malloc(100*sizeof(struct big));
```

4.1.4 RAM and ROM Models

The MP C compiler produces code that is suitable for use as firmware in a ROM-based system. In such a system, the initialization tables in the .cinit section are stored in ROM. At system initialization time, the C boot routine copies data from these tables from ROM to the initialized variables in .bss (RAM).

In situations where a program is loaded directly from an object file into memory and then run, you can avoid having the .cinit section occupy space in memory. A loader can read the initialization tables directly from the object file (instead of from ROM) and perform the initialization directly at load time (instead of at runtime). You can specify this *to the linker* by using the –cr linker option. For more information, see Section 4.8, *System Initialization*.

4.2 Maintaining Data Cache Coherency

The MP processor has a data cache that increases the overall performance of the processor, but to get the most performance from it, you must follow some special conventions when you are communicating with other processors through shared memory.

4.2.1 The Buffered Producer/Consumer Relationship

When the MP writes a stream of data that is to be shared, you must flush the data from the MP's data cache before having any other processor read the data. This coherency mechanism requires that all the data be written and flushed by the MP before another processor reads any of the data. You can use the MP run-time-support function flush() to flush the data after the MP has written it. Refer to Chapter 5, *Runtime-Support Functions* for more information on the flush() function.

Similarly, when the MP reads a stream of data that has been written by another processor, you must ensure that the MP will read the data which was written into external memory, and not a stale copy of the data in the MP's data cache. Therefore, you should flush the data area before the other processor writes any data into the area. The MP can then read the data after it has all been written by the other processor. The MP will be guaranteed to get the correct copy, because the flush ensured that a stale copy of the data was not present in cache.

4.2.2 The Unbuffered Producer/Consumer Relationship

When the MP needs to share data with another processor and buffering data is not an option or only a scalar is involved, then the MP's data cache can be bypassed. The MP supports special load and store instructions which interface directly with external memory. These instructions can be used to bypass the cache. Note that these cache bypass loads and stores are significantly less efficient than the corresponding cached loads and stores, so do not use them indiscriminately.

The cache bypass instructions present another problem, in addition to slower performance. You should not use a cache bypass load/store on an address in a cache subblock when you are referencing any other address in that cache subblock with a normal cached load/store. The following example illustrates the problem associated with doing so.

The problem in the example is that the cache flush overwrote the value 5 which was written by a cache bypass store. This problem

can be avoided if a cache subblock is never flushed when it contains an address that must be referenced with a cache bypass instruction.

Example 4–1. Cache Flush Overwrites Memory

	Cache 0x02000000	Cache 0x02000004	Mem 0x02000000	Mem 0x02000004
`move 5, r2`	XXX	XXX	XXX	XXX
`st 0x02000000(r0), r0`	0	XXX	XXX	XXX
`dst 0x02000004(r0), r2`	0	XXX	XXX	5
flush occurs	0	XXX	0	XXX

Note: dst indicates a cache bypass store

If you follow the convention that all addresses within a cache subblock will either exclusively be referenced with normal cached instructions or exclusively be referenced with cache bypass instructions, you can avoid this problem.

You can easily achieve this in C by placing all variables that are referenced with cache bypass instructions into a separate section and ensuring at link time that this section does not share a cache subblock with any other section. You can use the .usect directive to place variables into a specific section in assembly code. You can use the DATA_SECTION pragma in C code to specify that a variable is to be placed into a particular section. For more information on the DATA_SECTION pragma, see Section 2.8, *Pragma Directives*. For more information on ensuring that a section will not occupy a cache subblock with another section, see Chapter 14, *Linking PP and MP Files: An Extended Example*.

4.2.3 Cache Bypass Loads/Stores in C

Eleven macros are provided in the header file mvp.h that can be used to provide access to cache bypassed memory locations from C source. The macros are:

- NOCACHE_CHAR(x)
- NOCACHE_UCHAR(x)
- NOCACHE_SHORT(x)
- NOCACHE_USHORT(x)
- NOCACHE_INT(x)
- NOCACHE_UINT(x)
- NOCACHE_LONG(x)
- NOCACHE_ULONG(x)
- NOCACHE_FLOAT(x)
- NOCACHE_DOUBLE(x)
- NOCACHE_LDOUBLE(x)

These macros can be used on either side of an assignment as follows:

```
#include <mvp.h>
int x,y,z;

x = NOCACHE_INT(y);   ; cache bypass load of y
NOCACHE_INT(z) = x;   ; cache bypass store of z
```

Each macro is used to reference memory as the type indicated in the name of the macro, i.e. x = NOCACHE_INT(y) will load a signed 32-bit value at an address represented by y, and x = NOCACHE_USHORT(y) will load an unsigned 16-bit value at an address represented by y.

The macros take the address of their argument, so it is required that the argument be a valid value (i.e. something that is legal to assign to). For example:

```
NOCACHE_INT(variable)                ; ok
NOCACHE_INT(* (int *) 0x20001000)    ; ok
NOCACHE_INT((var1 + var2))           ; not ok
NOCACHE_INT((int) variable)          ; not ok
NOCACHE_INT(variable++)              ; not ok
```

Note: Use the Correct Argument Types

If you pass an argument with a different type from the type in the macro name, the amount of memory referenced and the result of the expression will be taken from the type in the name of the macro, not from the arguments type. For example, the code below:

```
float a, b;
a = NOCACHE_SHORT(b);
```

will load a two's-complement 16-bit value from the address that b represents, convert that value to float and assign the converted value to a.

4.3 Object Representation

This section explains how various data objects are sized, aligned, and accessed.

4.3.1 Data Type Storage

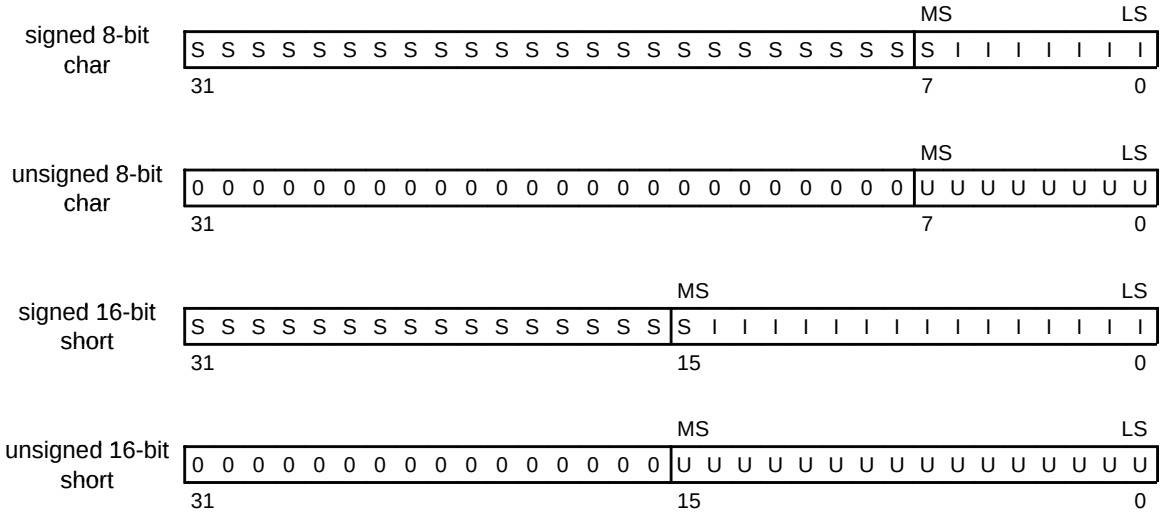
The following chart illustrates register and memory storage in the MP for various data types:

Table 4–1. MP Data Representation in Registers and Memory

Data Type	Register Storage	Memory Storage
char	bits 0–7 of register	1 byte
unsigned char	bits 0–7 of register	1 byte
short	bits 0–15 of register	2 bytes
unsigned short	bits 0–15 of register	2 bytes
int	bits 0–31 of register	4 bytes
unsigned int	bits 0–31 of register	4 bytes
enum	bits 0–31 of register	4 bytes
long	bits 0–31 of register	4 bytes
unsigned long	bits 0–31 of register	4 bytes
float	bits 0–31 of register	4 bytes
double	even/odd register pair	8 bytes
long double	even/odd register pair	8 bytes
struct	members stored as their individual types require	multiple of 4 bytes aligned to 64-bit boundary. Members stored as their individual types require
array	members stored as their individual types require	members stored as their individual types require

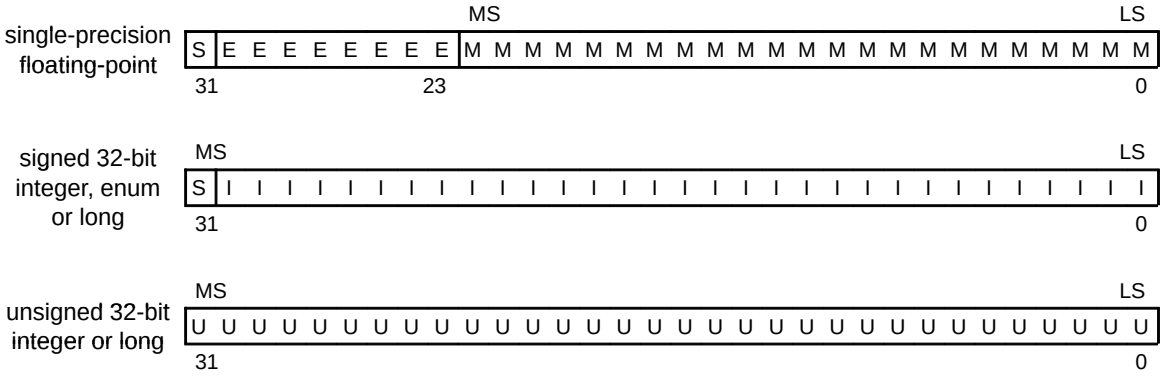
Char and unsigned char objects are stored in memory as a single byte, and are loaded to and stored from bits 0–7 of a register (see Figure 4–1). Objects defined as short or unsigned short are stored in memory as two bytes, and are loaded to and stored from bits 0–15 of a register (see Figure 4–1). In big-endian mode, 2-byte objects are loaded to registers by moving the first byte (that is, lower address) of memory to bits 8–15 of the register and moving the second byte of memory to bits 0–7. In little-endian mode, 2-byte objects are loaded to registers by moving the first byte (i.e., lower address) of memory to bits 0–7 of the register and moving the second byte of memory to bits 8–15.

Figure 4–1. Char and Short Data in MP Registers



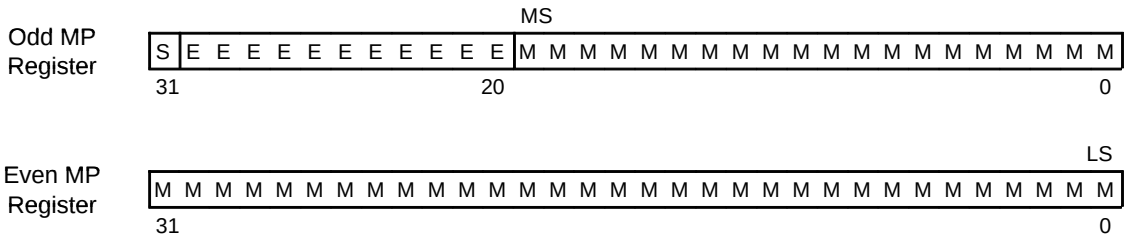
Six data types are stored in memory as 32-bit objects: int, unsigned int, enum, long, unsigned long, and float. Objects of these types are loaded to and stored from bits 0–32 of a register. In big-endian mode, four-byte objects are loaded to registers by moving the first byte (i.e., lower address) of memory to bits 24–31 of the register, moving the second byte of memory to bits 16–23, moving the third byte to bits 8–15, and moving the fourth byte to bits 0–7. In little-endian mode, four-byte objects are loaded to registers by moving the first byte (i.e., lower address) of memory to bits 0–7 of the register, moving the second byte to bits 8–15, moving the third byte to bits 16–23, and moving the fourth byte to bits 24–31.

Figure 4–2. 32-Bit Data in MP Registers



Double and long double values are stored in an even/odd pair of registers (see Figure 4–3) and are always referenced as an even register number. The odd memory word contains the sign bit, exponent, and the most significant part of the mantissa. The even memory word contains the least significant part of the mantissa.

Figure 4–3. Double-Precision Floating-Point Data in Register



A nested structure will only be aligned on a 4-byte boundary if it does not contain a double or a long double. Top level structures and nested structures containing a double or long double will be aligned on a 8-byte boundary. Structures will always reserve a multiple of four bytes of storage in memory unless the structure contains a double or a long double type; then the structure will reserve a multiple of eight bytes. Members of structures are stored in the same manner as if they were individual objects.

Arrays will be aligned on a boundary required by their element types. Elements of arrays are stored in the same manner as if they were individual objects.

4.3.2 MP Bit Fields

Bit fields are the only objects that are packed within a byte; that is, two bit fields may be stored in the same byte. Bit fields may range in size from 1 to 32 bits, but they will never span a 4-byte boundary.

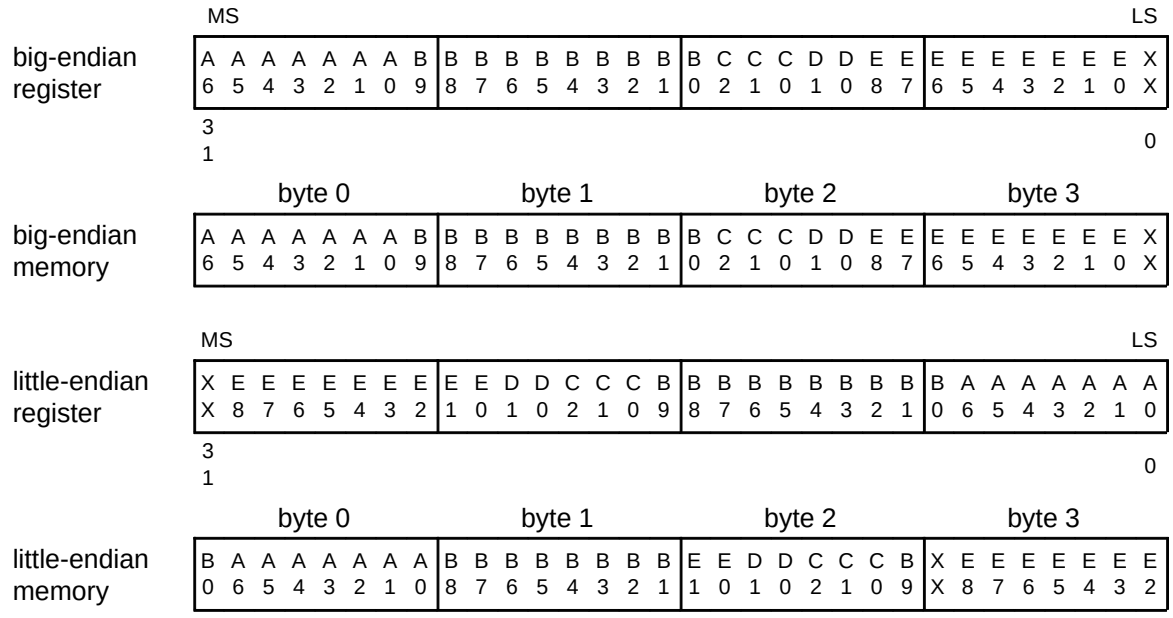
For big-endian mode, bit fields are packed into registers from MSB to LSB in the order in which they are declared, and packed in memory from MSbyte to LSbyte. (see Figure 4–4). For little-endian mode, bit fields are packed in registers from the LSB to the MSB in the order in which they are declared, and packed in memory from LSbyte to MSbyte (see Figure 4–4).

In the following example of bit field packing, assume these bit field declarations:

```
struct{
   int A:7
   int B:10
   int C:3
   int D:2
   int E:9
}x;
```

Note that A0 represents the least significant bit of the field A; A1 represents the next least significant bit, etc. Note again that memory storage for bit fields is done with a byte-by-byte, rather than bit-by-bit, transfer.

Figure 4–4. Bit Field Packing in Big-Endian and Little-Endian Formats



4.3.3 Character String Constants

In C, a character string constant can be used in one of two ways:

- It can initialize an array of characters; for example:

  ```
  char s[] = ″abc″;
  ```

  When a string is used as an initializer, it is simply treated as an initialized array; each character is a separate initializer. For more information about initialization, see Section 4.8, *System Initialization*.

- It can be used in an expression; for example:

  ```
  strcpy (s, ″abc″);
  ```

  When a string is used in an expression, the string itself is defined in in the .const section with the .string assembler directive, along with a unique label that points to the string; the terminating 0 byte is included. For example, the following lines define the string abc, and the terminating 0 byte (the label SL5 points to the string):

  ```
        .sect ″.const″
  SL5:  .string ″abc″,0
  ```

  String labels have the form **SL***n*, where *n* is a number assigned by the MP compiler to make the label unique. The number begins at 0 and is increased by 1 for each string defined. All strings used in a source module are defined at the end of the compiled assembly language module.

  The label SL*n* represents the address of the string constant. The MP compiler uses this label to reference the string expression.

  Because strings are stored in the .const section (possibly in ROM) and shared, it is bad practice for a program to modify a string constant. The following code is an example of incorrect string use:

  ```
  char  *a = ″abc″
  a[1] = ’x’;          /* Incorrect!  */
  ```

4.4 MP Register Conventions

Strict conventions associate specific registers with specific operations in the C environment. If you plan to interface an assembly language routine to a C program, it is important that you understand these register conventions.

The register conventions dictate how the MP compiler uses registers and how values are preserved across function calls. Table 4–2 summarizes how the MP compiler uses the TMS320C8x registers.

Table 4–2. Registers That May be Allocated for Variables and Expression Analysis

Register	Preserved By Call	Special Uses
r0	—	Zero Register
r1	no	Stack Pointer
r2	no	Return and Argument 1
r3	no	Return and Argument 1 (with r2 for doubles/ long doubles)
r4	no	Argument 2
r5	no	Argument 2 (with r4 for doubles/long doubles)
r6	no	Argument 3
r7	no	Argument 3 (with r6 for doubles/long doubles)
r8	no	Argument 4
r9	no	Argument 4 (with r8 for doubles/long doubles)
r10	no	Argument 5
r11	no	Argument 5 (with r10 for doubles/long doubles)
r12	no	Argument 6
r13	no	Argument 6 (with r12 for doubles/long doubles)

Register	Preserved By Call	Register	Preserved By Call
r14	no	r15	no
r16	no	r17	no
r18	no	r19	no
r20	yes	r21	yes
r22	yes	r23	yes
r24	yes	r25	yes
r26	yes	r27	yes
r28	yes	r29	yes
r30	yes	r31 (Link Register)	yes

When a function call occurs, the called function is responsible for preserving the contents of certain registers. The caller need not save the contents of these registers. If the caller is using a register that is not preserved, the caller must save the register's value before making the call.

4.4.1 Register Variables and Register Allocation

The registers in Table 4–2 are available to the MP compiler for allocation to register variables and temporary expression results. If the MP compiler cannot allocate a register of a required type, then spilling will occur. Spilling is the process of moving a register's contents to memory to free the register for another purpose.

Objects of type double or long double will be allocated into an even/odd register pair and are always referenced as an even register number. The odd register contains the sign bit, exponent, and the most significant part of the mantissa. The even register contains the least significant part of the mantissa. The r3 register is used with r2 for passing the first argument, if the first argument is a double or long double. The same is true for r4 and r5 for the second parameter, and so on. For more information about arguments passing registers and return registers, see Section 4.5, *Function Structure and Calling Conventions.*

4.4.2 Control Registers

You can use the cregister keyword to access the control registers from C. If you use one of these registers in C code and the register may be changed by an external mechanism, then you will want to declare the register to be volatile. For example, to read the INTPEN register:

```
cregister extern volatile unsigned int INTPEN;
main()
{
     unsigned int x = INTPEN;
}
```

The cregister keyword is only allowed on declarations that occur outside of a function, and the type of the variable must be integer.

Table 4–3. Valid Control Registers for the MP

ANASTAT	BRK1	BRK2	CONFIG	DLRU
DTAG0	DTAG1	DTAG2	DTAG3	DTAG4
DTAG5	DTAG6	DTAG7	DTAG8	DTAG9
DTAG10	DTAG11	DTAG12	DTAG13	DTAG14
DTAG15	ECOMCNTL	EIP	EPC	FLTADR
FLTDTH	FLTDTL	FLTOP	FLTTAG	FPST
IE	ILRU	IN0P	IN1P	INTPEN
ITAG0	ITAG1	ITAG2	ITAG3	ITAG4
ITAG5	ITAG6	ITAG7	ITAG8	ITAG9
ITAG10	ITAG11	ITAG12	ITAG13	ITAG14
ITAG15	MIP	MPC	OUTP	PKTREQ
PPERROR	SYSSTK	SYSTMP	TCOUNT	TSCALE

The MP control registers are declared in the header file mvp.h which is shipped with the compiler. For more information, see Section 2.5, *The cregister Keyword*.

4.5 Function Structure and Calling Conventions

The MP C compiler imposes a strict set of rules on function calls. Except for special runtime-support functions, any function that calls or is called by a C function must follow these rules. Failure to adhere to these rules can disrupt the C environment and cause a program to fail.

4.5.1 Responsibilities of the Calling Function

A function performs the following tasks when it calls another function.

1) If any arguments are to be passed to a function, then as many as the first six arguments are placed in registers. With the exception of doubles and long doubles, the arguments will be passed in registers r2, r4, r6, r8, r10, and r12. If a double or long double is passed, then they will be passed in a register pair, consisting of one of the even register numbers from above along with the odd register number that follows it.

   Any remaining arguments are placed on the stack in reverse order (that is, the stack pointer points to the seventh argument; SP+*offset* points to the eighth argument, etc.) Remember that arguments placed on the stack must be aligned to a value appropriate for their size, and also that any argument that is not defined in a prototype and whose size is less than an int's size will be passed as an int.

   A structure argument will be passed as the address of the structure. It will be up to the called function to make a local copy.

   If a function is declared with a ellipse indicating that it can be called with varying numbers of arguments,then the convention is slightly modified. The last explicitly declared argument is passed on the stack, so that its stack address can act as a reference for accessing the undeclared arguments.

Figure 4–5. Register Argument Conventions

```
int func1(int a, int b, int c);
          r2      r4      r6

int func2(int a, float b, int *c, struct A d, float e, int f, int g);
          r2       r4       r6       r8          r10      r12     stack

int func3(int a, double b, float c, long double d);
           r2      r4:r5       r6           r8:r9

int vararg(int a, int b, int c, ...);
          r2       r4      stack   stack    stack    stack...
```

The called function is responsible for preserving r20–r31. Any other registers whose contents must be preserved must be pushed onto the stack.

2) The caller calls the function.

3) The caller reclaims any stack space needed for arguments by adding to the stack pointer. This step is only needed in assembly programs that were not compiled from C code, because the MP C compiler allocates the stack space needed for all calls at the beginning of the function, and deallocates at the end of the function.

4.5.2 Responsibilities of a Called Function

A called function must perform the following tasks.

1) The called C function will allocate enough space on the stack for any local variables, temporary storage areas, and arguments to functions that this function may call. This allocation occurs once at the beginning of the function by subtracting a constant from the SP register. The SP register will not be decremented anywhere else within this function.

2) If the called function calls any other functions, then the return address must be saved on the stack. Otherwise, it is left in the link register R31 and will be overwritten by the next function call.

3) If the called function modifies any save-on-entry registers (r20–r31), it must save them, either in other registers or on the stack. The called function may modify any other registers without saving them.

4) If the called function is expecting a structure argument, it will be passed a pointer to the structure instead. If any writes are made to the structure from within the called function, then space for a local copy of the structure must be allocated on the stack, and the local structure must be copied from the passed pointer to the structure. If no writes are made to the structure, then it can be referenced in the called function indirectly through the pointer argument.

   You must be careful to properly declare functions that accept structure arguments both at the point where they are called (so that the structure argument is passed as an address) and at the point where they are defined (so the function knows to copy the structure to a local copy).

5) The called function will execute the code for the function.

6) If the called function returns any integer, pointer or the float type, then the return value will be placed in the r2 register. If the function returns a double or long double type, the value will be placed in the r2:r3 register pair.

   If the function returns a structure, the caller allocates space for the structure and then passes the address of the return space to the called function as the first argument. To return a structure, the called function copies the structure to the memory block pointed to by the extra argument.

   In this way, the caller can be smart about telling the called function where to return the structure. For example, in the statement s = f(x), where s is a structure and f is a function that returns a structure, the caller can actually make the call as f(&s, x). The function f then copies the return structure directly into s, performing the assignment automatically.

   If the caller does not use the return structure value, then an address value of 0 can be passed as the first argument. This directs the called function not to copy the return structure.

   You must be careful to properly declare functions that return structures both at the point where they are called (so that the extra argument is passed) and at the point where they are defined (so the function knows to copy the result).

7) Any save-on-entry register which was saved in step 3, will be restored.

8) The space allocated for the function in step 1 will be reclaimed at the end of the function by adding a constant to the SP register.

9) The function returns by jumping to the value of link register R31 or the saved value of the link register.

4.5.3 Accessing Arguments and Local Variables

A function accesses its stack arguments and local nonregister variables indirectly through the stack pointer (SP), which always points to the top of the stack. Since the stack grows toward smaller addresses, the local and argument data for a function will be accessed with a positive offset from the SP register. Local variables, temporary storage, and the area reserved for stack arguments to functions called by this function will be accessed with offsets smaller than the constant subtracted from the SP at the beginning of the function.

Stack arguments passed to this function will be accessed with offsets greater than or equal to the constant subtracted from the SP at the beginning of the function. The MP compiler attempts to keep register arguments in their original registers if the optimizer is used or if they are declared with the register keyword. Otherwise, the arguments are copied to the stack to free those registers for further allocation.

4.6 Interfacing MP C With Assembly Language

There are two ways to use assembly language with C code:

- ❑ Use separate modules of assembled code and link them with compiled C modules (subsection 4.6.1, following). This is the most versatile method.
- ❑ Use inline assembly language, embedded directly in the C source (subsection 4.6.3, *Inline Assembly Language*).

4.6.1 Using Assembly Language Modules With C Code

Interfacing with assembly language functions is straightforward if you follow the calling conventions defined in Section 4.5, *Function Structure and Calling Conventions*, and the register conventions defined in Section 4.4, *MP Register Conventions*. C code can access variables and call functions defined in assembly language, and assembly code can access C variables and call C functions. Follow these guidelines to interface assembly language and C:

- ❑ All functions, whether they are written in C or assembly language, must follow the conventions outlined in Section 4.4, *MP Register Conventions*.
- ❑ You must preserve any dedicated registers modified by the function. The dedicated registers are r20—r31, and SP (r1). If you use the stack normally, you don't need to explicitly preserve the stack. In other words, you are free to use the stack inside a function as long as you pop everything you pushed before your function exits. You may use all other registers freely without preserving their contents.
- ❑ Interrupt routines must save *all* the registers they use. For more information about interrupt handling, see Section 4.7, *Interrupt Handling*.
- ❑ When you call a C function from assembly language, load the designated registers with arguments and push the remaining arguments onto the stack as described in subsection 4.5.2, *Responsibilities of a Called Function*. When you call a C function, remember that only r20–r31 and the stack pointer are preserved by the C compiler. C functions can alter any other registers, save any other registers whose contents need to be preserved by pushing them onto the stack before the function is called and restore them after the function returns.

- Functions must return values correctly according to their C declarations. Integers and 32-bit floating-point (float) values are returned in r2. Doubles and long doubles are returned in r2:r3, and structures are returned as described in subsection 4.5.2, *Responsibilities of a Called Function*.

- No assembly module should use the .cinit section for any purpose other than autoinitialization of global variables. The C startup routine in boot.asm assumes that the .cinit section consists *entirely* of initialization tables. Disrupting the tables by putting other information in .cinit can cause unpredictable results.

- The MP compiler appends an underscore (_) to the beginning of all identifiers. In assembly language modules, you must use an underscore prefix for all objects that are to be accessible from C. For example, a C object named x is called _x in assembly language. For identifiers that are to be used only in an assembly language module or modules, any name that does not begin with a leading underscore may be safely used without conflicting with a C identifier.

- Any object or function declared in assembly that is to be accessed or called from C must be declared with the .global or .sysmem directive in the assembler. This defines the symbol as external and allows the linker to resolve references to it.

  Likewise, to access a C function or object from assembly language, declare the C object with .global or .sysmem. This creates an undefined external reference that the linker will resolve.

Example 4–2 illustrates a C function called main, which calls an assembly language function called asmfunc. The asmfunc function takes its single argument, adds it to the C global variable called gvar, and returns the result.

Example 4–2. Calling An Assembly Language Function From C

C program

```
extern int asmfunc(); /* declare external asm function */
int gvar = 4;         /* define global variable        */

main()
{
   int i;
   i = 1;
   i = asmfunc(i);   /* call function normally        */
}
```

Assembly Language program (MP)

```
   .global   _gvar         ; declare external variables
   .global   _asmfunc      ; declare external function
_asmfunc:
             ld     _gvar (r0),r3
             jsr    r31(r0),0
             addu   r2,r3,r2
```

In the C program in Example 4–2, the extern declaration of asmfunc is optional because the return type is int. Like C functions, assembly functions need to be declared only if they return nonintegers.

4.6.2 Accessing Assembly Language Variables From C

It is sometimes useful for a C program to access variables defined in assembly language. There are three methods that you can use to accomplish this, depending on where and how the item is defined: a variable defined in the .bss section, a variable not defined in the .bss section, or a constant.

Accessing assembly language variables from C

Accessing uninitialized variables from the .bss section or a section named with .usect is straightforward:

1) Use the .bss or .usect directive to define the variable.

2) Use the .global directive to make the definition external.

3) Remember to precede the name with an underscore in assembly language.

4) In C, declare the variable as *extern*, and access it normally.

Example 4–3 shows an example that accesses a variable defined in .bss.

Example 4–3. Accessing a Variable From C

Assembly Language program (MP)

```
* Note the use of underscores in the
*         following lines

   .bss      _var,4,4  ; Define the variable
   .global   _var      ; Declare it as external
```

C program

```
extern int var;     /* External variable       */
var = 1;            /* Use the variable        */
```

Accessing assembly language constants

You can define global constants in assembly language by using the .set, .system, and .global directives, or you can define them in a linker command file using a linker assignment statement. These constants are accessible from C only with the use of special operators.

For normal variables defined in C or assembly language, the symbol table contains the *address of the value* of the variable. For assembler constants, however, the symbol table contains the *value* of the constant. The MP compiler cannot tell which items in the symbol table are values and which are addresses.

If you try to access an assembler (or linker) constant by name, the MP compiler will attempt to fetch a value from the address represented in the symbol table. To prevent this unwanted fetch, you must use the & (address of) operator to get the value. In other words, if x is an assembly language constant, its value in C is &x.

You can use casts and #defines to ease the use of these symbols in your program, as in Example 4–4.

Example 4–4. Accessing an Assembly Language Constant From C

Assembly Language program (MP)

```
_table_size   .set   10000         ; define the constant
              .global _table_size ; make it global
```

C program

```
extern int table_size; /*external ref */

#define TABLE_SIZE ((int) (&table_size))

       .         /* use cast to hide address-of */
       .
       .

for (i-0; i<TABLE_SIZE; ++i)

                     /* use like normal symbol */
```

Since you are referencing only the symbol's value as stored in the symbol table, the symbol's declared type is unimportant. In Example 4–4, int is used. You can reference linker-defined symbols in a similar manner.

4.6.3 Inline Assembly Language

Within a C program, you can use the **asm statement** to inject a single line of assembly language into the assembly language file that the MP compiler creates. A series of asm statements places sequential lines of assembly language into the MP compiler output with no intervening code. For more information about the asm statement, see Section 2.9, *The asm Statement*.

Note: Using the MP asm Statement

The asm statement is provided so you can access features of the hardware that would be otherwise inaccessible from C. When you use the asm statement, be extremely careful not to disrupt the C environment. The MP compiler does not check or analyze the inserted instructions.

Do not use the asm statement to insert assembler directives that would change the assembly environment.

The asm statement is also useful for inserting comments in the MP compiler output; simply start the assembly code string with a semicolon (;) as shown below:

```
asm(";*** this is an assembly language comment");
```

4.7 Interrupt Handling

As long as you follow the guidelines in this section, C code can be interrupted without disrupting the C environment. When the C environment is initialized, the startup routine does not enable or disable interrupts. If your system uses interrupts, it is your responsibility to handle any required enabling or masking of interrupts. Such operations have no affect on the C environment and can be easily incorporated with *asm* statements.

4.7.1 Saving Registers During Interrupts

When C code is interrupted, the interrupt routine must preserve the contents of all machine registers that are used by the routine or by any function called by the routine.

The compiler will handle register preservation if the interrupt service routine is written in C.

4.7.2 Using MP C Interrupt Routines

Interrupt routines can be declared in C with the interrupt keyword in the function definition. Interrupt functions must be defined as returning void type, and may not have any arguments. For example:

```
interrupt void example ()
{
...
}
```

The name c_int00 is the C entry point; this name is reserved for the system reset interrupt. This special interrupt routine initializes the system and calls the function main. Because it has no caller c_int00 does not save any registers.

If a C interrupt routine does not call any other functions, only those registers that are actually used in the interrupt handler are saved and restored. However, if a C interrupt routine *does* call other functions, these functions may modify unknown registers that are not used in the interrupt handler itself. For this reason, the routine saves all usable registers if any other functions are called.

Interrupt handling functions should not be called directly. For more information, see Section 2.6, *The interrupt and trap Keywords*.

4.7.3 Assembly Language Interrupt Routines

You can handle interrupts with assembly language code, as long as you follow the register conventions as the compiler does. Like *all* assembly functions, interrupt routines can use the stack, access global C variables, and call C functions normally. When calling C functions, be sure that any register listed in Table 4–2, is saved, because the C function can modify them. The PP compiler will preserve the values in d6, d7, a4, a12, and SP, so these registers need not be saved explicitly in assembly.

4.8 System Initialization

Before you can run a C program, the C runtime environment must be created. This task is performed by the C boot routine, which is a function called c_int00.

The c_int00 function can be branched to, called, or vectored by reset hardware to begin running the system. The function is in the runtime-support library and must be linked in with the other C object modules. This occurs automatically when you use the –c or –cr option in the linker and include a runtime-support library created from mp_rts.src as one of the linker input files. When C programs are linked, the linker sets the entry point value in the executable output module to the symbol c_int00.

The c_int00 function performs the following tasks to initialize the environment:

1) It defines a section called .stack for the system stack and sets up the initial stack and frame pointers.

2) It autoinitializes global variables by copying the data from the initialization tables in .cinit to the storage allocated for the variables in .bss. In the RAM initialization model, a loader performs this step before the program runs (it is not performed by the boot routine). For more information, see subsection 4.8.1.

3) It calls the function main to begin running the C program.

You can replace or modify the boot routine to meet your system requirements. However, the boot routine *must* perform the three operations listed above to correctly initialize the C environment. The runtime-support source library contains the source to this routine in a module named boot.asm.

4.8.1 MP Autoinitialization of Variables and Constants

Some global variables must have initial values assigned to them before a C program starts running. The process of retrieving these variables' data and intializing the variables with the data is called autoinltialization.

The MP compiler builds tables in a special section called **.cinlt** that contains data for initializing global and static variables. Each compiled module contains these initialization tables. The linker combines them into a single table (a single .cinit section). The boot routine uses this table to initialize all the variables that need values before the program starts running.

Note: Initializing Variables

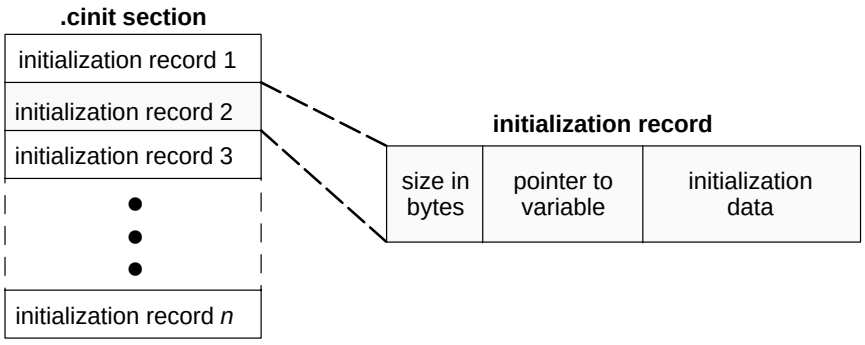
In standard C, global and static variables that are not explicitly initialized are set to 0 before program execution. The MVP MP compiler does not perform any preinitialization of uninitialized variables. Any variable that must have an initial value of 0 must be explicitly initialized. An alternative is to have a loader or boot.obj clear the .bss section before the program starts **running.**

There are two methods for copying the initialization data into memory: RAM and ROM. The RAM model of initialization is discussed in *Initializing variables in the RAM model for the MP*, page CG: 4-31, and *ROM initialization model for the MP*, page CG: 4-32, describes the ROM model of initialization.

Initialization tables

The tables in the .cinit section consist of initialization records with varying sizes. Each initialization record has the following format:

Figure 4–6. Format of Initialization Records in the .cinit Section



- The first field (word 0) is the size in bytes of the initialization data for the variable.
- The second field (word 1) is the starting address of the area where the initialization data must be copied (this field points to a variable).
- The first two fields are followed by one or more bytes of data. During autoinitialization, this data is copied to the specified address.

The .cinit section contains an initialization record for each variable that must be initialized. Example 4–5 shows initialized variables defined in C:

Example 4–5. MP Initialization Table

```
int x;
int i = 23;
int *p = &x;
int a[5] = {1,2,3,4,5}
```

The initialization information for these variables is:

```
   .sect  ".cinit"
   .align 4          ; each record aligned on 4-byte boundary
   .field 4,32       ; length of data is 4 bytes
   .field _i+0,32    ; address of i
   .field 23,32      ; data is 23

   .sect  ".cinit"
   .align 4
   .field 4,32       ; length of data is 4 bytes
   .field _p+0,32    ; initialize p
   .field _x,32      ; data is value of x

   .sect  ".cinit"
   .align 4
   .field IR1,32     ; length is 20
   .field _a+0,32    ; address is a
   .field 1,32       ; _a[0] data is 1
   .field 2,32       ; _a[1] data is 2
   .field 3,32       ; _a[2] data is 3
   .field 4,32       ; _a[3] data is 4
   .field 5,32       ; _a[4] data is 5
IR1:   .set   20
```

The .cinit section contains only initialization tables in this format. When interfacing assembly language modules, do not use the .cinit section for any other use.

When you link a program with the –c or –cr option, the linker links together the .cinit sections from all the C modules and appends a null word to the end of the entire section. This appears as a record with a size field of 0 and marks the end of the initialization tables.

Initializing variables in the RAM model for the MP

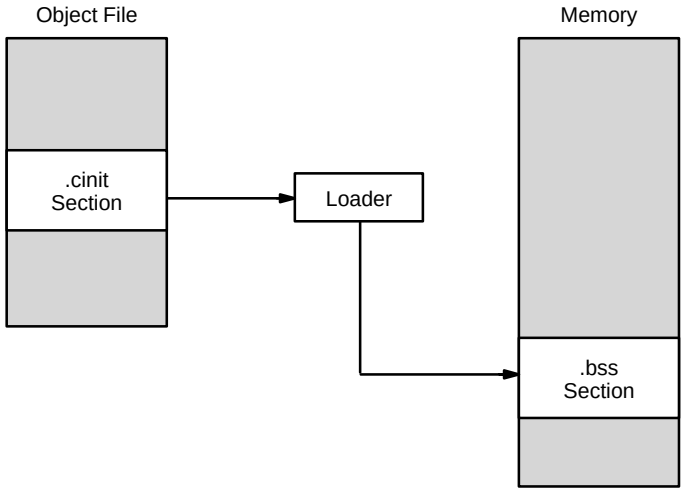
The RAM model, specified with the –cr linker option, allows variables to be initialized at loadtime instead of at runtime. This can enhance performance by reducing boot time and can save the memory used by the initialization tables. The RAM option requires the use of a smart loader to perform the initialization as it copies the program from the object file into memory.

In the RAM model, the linker marks the .cinit section with a special attribute. This means that the section is *not* loaded into memory and does *not* occupy space in the memory map. The symbol cinit is set to –1 to indicate to the C boot routine that the initialization tables are not present in memory; accordingly, no runtime initialization is performed at boot time.

Instead, when the program is loaded into memory, the loader must detect the presence of the .cinit section and its special attribute. Instead of loading the section into memory, the loader uses the initialization tables directly from the object file to initialize the variables. To use the RAM model, the loader must understand the format of the initialization tables so that it can use them.

A loader is not part of the MP compiler package.

Figure 4–7. RAM Model of Autoinitialization

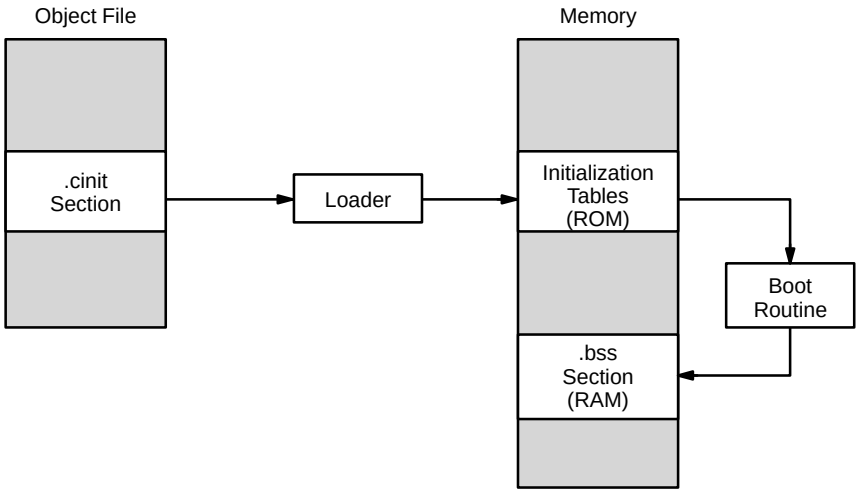


ROM initialization model for the MP

The ROM model is the default model for autoinitialization. To use the ROM model, invoke the linker with the –c option.

Under this method, the .cinit section is loaded into memory possibly ROM) along with all the other sections, and global variables are initialized at *runtime*. The linker defines a special symbol called .cinit that points to the beginning of the tables in memory. When the program begins running, the C boot routine copies data from the tables (pointed to by .cinit) into the specified variables. This allows initialization data to be stored in ROM and then copied to RAM each time the program is started.

Figure 4–8. ROM Model of Autoinitialization



Chapter 5

Runtime-Support Functions

Some of the tasks that a C program may need to perform (such as I/O, dynamic memory allocation, string operations, and trigonometric functions) are not part of the C language itself. However, the ANSI C standard defines a set of runtime-support functions that perform these tasks. The TMS320C8x C compilers include a set of libraries that implement the complete ANSI standard library except for those facilities for handling exception conditions and locale issues (properties that depend on local language, nationality, or culture). Using the ANSI standard library ensures a consistent set of functions that provide for greater portability.

In addition to the ANSI-specified functions, the TMS320C8x runtime-support libraries include routines that allow a user to:

- have direct access to specific MVP master processor commands
- direct C language I/O requests to a user-specified device

A library build utility is provided with the code generation tools that lets you create customized runtime-support libraries. The use of this utility is covered in Chapter 6, *Library Build Utility*.

Topics

5.1 Libraries

Eight object libraries and two source libraries are included with the TMS320C8x C compiler:

C I/O Object Libraries

mp_cio.lib	MP ANSI standard I/O — big endian
mp_ciol.lib	MP ANSI standard I/O — little endian
pp_cio.lib	PP ANSI standard I/O — big endian
pp_ciol.lib	PP ANSI standard I/O — little endian

Runtime-support Object Libraries

mp_rts.lib	MP runtime-support functions — big endian
mp_rtsl.lib	MP runtime-support functions — little endian
pp_rts.lib	PP runtime-support functions — big endian
pp_rtsl.lib	PP runtime-support functions — little endian

Source Libraries

mp_rts.src	Source libraries for mp_rts[l].lib
pp_rts.src	Source libraries for pp_rts[l].lib

The ANSI C standard library has been divided into two object libraries. The C I/O library implements all the functions necessary to perform I/O between the MVP and the host operating system on which the debugging tools are being executed. It contains all the functions declared in the header file stdio.h. The runtime-support library contains the remaining functions defined in the ANSI C standard library except for those involving signals and locale issues. The C I/O and runtime-support libraries have been compiled for both the MP and PP processors operating in either big or little endian mode.

In addition to the ANSI standard library, the C I/O and runtime-support object libraries include:

- The intrinsic arithmetic routines described in Section 3.7, *Runtime-Support Arithmetic Routines*.
- The system startup routines, _c_int00 (MP) and $c_int00 (PP).
- Low level support functions that provide I/O to the host OS
- Functions that allow I/O between the MVP and a user-added device.
- Functions and macros that allow C to access specific MP instructions.

Included with the object libraries are the source libraries mp_rts.src and pp_rts.src. The runtime-support object libraries are built from the C and assembly source contained in *xx*_ rts.src. The C I/O source code is not included with the TMS320C8X C code generation tools.

5.1.1 Linking Code with the Object Libraries

When you link your program, you must specify an object library as one of the linker input files so that references to either a C input/output or runtime-support functions can be resolved. You should usually specify libraries *last* on the linker command line because the linker searches a library for unresolved references when it encounters the library on the command line. You can also use the –x linker option to force repeated searches of each library until the linker can resolve no more references. When a library is linked, the linker includes only those library members required to resolve undefined references. For more information about linking, see Section 13.15, *Linking C Code*.

Note: Duplicate exit Functions Exist

Both the C I/O and runtime-support libraries have definitions for the exit function which is called at the termination of any C program. The exit function defined in the C I/O lib performs extra bookkeeping tasks required when performing I/O. When using any C I/O function a user must link in the C I/O version of exit and not the one that is defined in the runtime-support library. When linking both libraries use –u _exit or (–u \$exit for the PP) on the linker command line before linking in either library, and link in the C I/O library before the runtime-support library. For example:

```
mvplnk –c file.obj –u _exit –l mp_cio.lib –l mp_rts.lib
–o file.out
```

would ensure the exit function defined in mp_cio.lib would be linked into the final object file.

When using the –u option to undefine $exit for the PP, prefix the dollar sign ($) with a back slash so that the dollar sign is not interpreted by the OS.

```
mvplnk –pc file.obj –u \$exit ...
```

5.1.2 Modifying a Library Function

You can inspect or modify library functions by using the archiver to extract the appropriate source file or files from the source libraries. For example, the following command extracts two source files:

```
mvpar x pp_rts.src atoi.c strcpy.c
```

To modify a function, extract the source as in the previous example. Make the required changes to the code, recompile, and then reinstall the new object file or files into the library:

```
ppcl –options atoi.c strcpy.c            ;recompile
mvpar r pp_rts.lib  atoi.obj strcpy.obj  ;rebuild library
```

You can also build a new library this way, rather than rebuilding back into pp_rts.lib. For more information about the archiver, see Chapter 16, *Archiver Description*.

5.1.3 Building a Library With Different Options

You can create a new library from mp_rts.src or pp_rts.src by using the library-build utilities, mpmk and ppmk. For example, use this command to build an optimized runtime-support library for the TMS320C8x MP:

```
mpmk ––u –o2 –x mp_rts.src –1 mp_rts2.lib
```

The ––u option tells the mpmk utility to use the header files in the current directory, rather than extracting them from the source archive. The new library is compatible with any code compiled for the ’C8x MP. The use of the optimizer (–o2) and inline function expansion (–x) options does not affect compatibility with code compiled without these options. For more information on the library build utilities, see Chapter 6, *Library Build Utility*.

5.2 The C I/O Library

The C I/O library makes it possible for the MVP to access the host's OS to perform I/O (via the debugger). For example, printf statements executed in an MVP program will appear in the debugger's command window. When used in conjunction with the MVP debugging tools the capability to perform I/O on the host allows users more options when debugging and testing code written for the MVP.

To use the C I/O library:

- Include the header file stdio.h for each module that references a function in the C I/O library.
- Allow for 320 bytes of heap space for each I/O stream used in your program. A stream is a source or destination of data that is associated with a peripheral, such as a terminal or keyboard. Streams are buffered using dynamically allocated memory which is taken from the heap. More heap space may be required to support programs which use additional amounts of dynamically allocated memory (calls to malloc()). To set the heap size, use the –heap or –pheap options when linking. See Sections 13.3.6, *Define MP Heap Size (–heap* size *Option)* and 13.3.10, *Define PP Heap Size (–pheap* size *Option)*, for more information.
- Link in both the C I/O and runtime-support libraries appropriate for the desired processor and endianess. As mentioned in section 5.1 link the C I/O library before the runtime-support library and use either the –u _exit (MP) or –u $exit (PP) linker option before linking in either library. This ensures that a C program will terminate properly when using C I/O.
- Invoke the debugger with the –o option to enable C I/O support. For more information about the debugger's –o option, see page DB:1-18, *Enabling C I/O functions (–o option)*.

For example, given the following program in a file named main.c:

```
#include <stdio.h>

main()
{
 FILE *fid;

   fid = fopen(”myfile”,”w”);
   fprintf(fid,”Hello, world\n”);
   fclose(fid);

   printf(”Hello again, world\n”);
}
```

Issuing the MP shell command:

```
mpcl main.c –z –heap 400 –u _exit –l mp_cio.lib –l mp_rts.lib –o main.out
```

will compile, link and create a file main.out. Executing main.out under the MVP debugger on a SPARC host will:

1) Open the file *myfile* in the directory where the debugger was invoked.

2) Print the string *Hello, world* into that file

3) Close the file

4) Print the string *Hello again, world* in the debugger's command window

Note: C I/O Can Only Be Used On a Single Processor

The interface between the debugger and the host OS and the common data structures used by C I/O make it impossible for you to use C I/O on more than one processor per application.

With properly written device drivers the C I/O library also offers facilities to perform I/O on a user-specified device.

5.2.1 Overview Of C I/O Implementation

The code that implements I/O is logically divided into 3 layers: high level, low level, and device level.

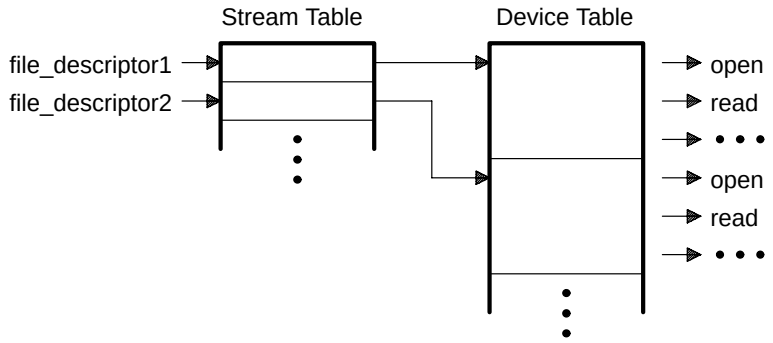
The high level functions are the standard C library of stream I/O routines visible to the user (printf, scanf, fopen, getchar, etc.). These routines map an I/O request to one or more of seven I/O commands that are handled by the low level routines.

The low level routines are comprised of seven basic I/O functions: OPEN, READ, WRITE, CLOSE, LSEEK, RENAME,and UNLINK. These low level routines provide the interface between the high level functions and the device level drivers which actually perform the I/O command on the specified device.

The low level functions also define and maintain a stream table which is used to associate a file descriptor with a device. The stream table interacts with the device table to ensure that an I/O command performed on a stream executes the correct device level routine.

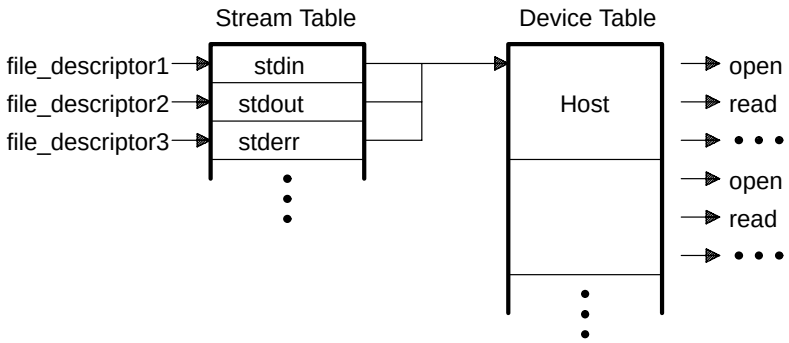
The data structures interact as shown in Figure 5–1.

Figure 5–1. Interaction of Data Structures in I/O Functions



The first three streams in the stream table are predefined to be stdin, stdout, and stderr and they point to the host device and associated device drivers.

Figure 5–2. The First Three Streams in the Stream Table



At the next level are the seven user-definable device level drivers. They map directly to the seven low level I/O functions. The C I/O library includes the device drivers necessary to perform C I/O on the host on which the debugger is running.

The specifications for writing device level routines so that they interface with the low level routines follow. Each function should set up and maintain its own data structures as needed. For certain devices, some function definitions perform no action and should just return.

CLOSE — Close File or Device For I/O

Syntax **int CLOSE(int** *file_descriptor***);**

Description Close the device or file associated with *file_descriptor*.

Parameters

file_descriptor is the stream number assigned by the low level routines that is associated with the opened device or file.

Return Value

–1 on error
0 if successful

LSEEK — Set File Position Indicator

Syntax **long LSEEK(int** *file_descriptor***, long** *offset***, int** *origin***);**

Description Sets the file position indicator for the given file to *origin* + *offset*. The file position indicator measures the position in characters from the beginning of the file.

Parameters

file_descriptor is the stream number assigned by the low level routines that the device level driver should associate with the opened file or device.

offset is used to indicate the relative offset from the *origin* in characters.

origin is the argument used to indicate which of the base locations the *offset* is measured from. The *origin* must be a value returned by one of the following macros:

SEEK_SET (0x0000) beginning of file
SEEK_CUR (0x0001) current value of the file position indicator
SEEK_END (0x0002) end of file

Return Value

EOF if unsuccessful
The new value of file position indicator if successful

OPEN — Open File or Device For I/O

Syntax

int OPEN(char *****path*, unsigned** *flags*, **int** *file_descriptor*);**

Description

Open the device or file specified by *path* preparing it for I/O and associate the device or file with the *file_descriptor*.

Parameters

path — is the filename of the file to be opened, including path information

flags — attributes of how file is to be manipulated. Specified using the following symbols:

```
O_RDONLY  (0x0000)  /* open for reading */
O_WRONLY  (0x0001)  /* open for writing */
O_RDWR    (0x0002)  /* open for read & write */
O_APPEND  (0x0008)  /* append on each write */
O_CREAT   (0x0100)  /* open with file create */
O_TRUNC   (0x0200)  /* open with truncation */
O_BINARY  (0x8000)  /* open in binary mode */
```

These parameters can be ignored in some cases, depending on how data is interpreted by the device. Note, however, that the high level I/O calls will look at how the file was opened in an fopen statement and will prevent certain actions depending on the open attributes.

file_descriptor — is the stream number assigned by the low level routines that the device level driver should associate with the opened file or device.

Return Value

< 0 on error
>= 0 if successful

READ — Read Characters From Buffer

Syntax **int READ(int** *file_descriptor***, char** **buffer***, unsigned** *count***);**

Description Reads the number of characters specified by *count* to the *buffer* from the device or file associated with *file_descriptor*.

Parameters

file_descriptor	is the stream number assigned by the low level routines that is associated with the opened file or device.
buffer	is the location of the buffer where the read characters will be placed.
count	is the number of characters to be read from the device or file.

Return Value
–1 on error
0 if EOF was encountered before the read was complete
Number of characters read in every other instance

RENAME — Rename File

Syntax **int RENAME(char ***_old_name_**, char ***_new_name_**);**

Description Changes the name of a file.

Parameters

old_name	is the current name of the file.
new_name	is the new name for the file

Return Value
0 if the rename is successful
non 0 if it fails

UNLINK — Delete File

Syntax **int UNLINK(char ***_path_**);**

Description Deletes the file specified by *path*.

Parameters

path	is the path name of the file to delete.

Return Value
0 if the file was successfully deleted
–1 if it failed.

WRITE — Write Characters to Buffer

Syntax

int WRITE(int *file_descriptor***, char ****buffer***, unsigned** *count***);**

Description

Writes the number of characters specified by *count* from the *buffer* to the device or file associated with *file_descriptor*.

Parameters

file_descriptor	is the stream number assigned by the low level routines that is associated with the opened file or device.
buffer	is the location of the buffer the write characters are stored in.
count	is the number of characters to be written to the device or file.

Return Value

–1 on error
Number of characters written in every other case.

5.2.2 Adding A Device For C I/O

The low level functions provide facilities that allow a user to add and use a device for I/O at runtime. The procedure for using these facilities is:

1) Define the device level functions as described in subsection 5.2.1.

> **Note: Use Unique Function Names**
>
> The function names open(), close(), read(), etc. have been used by the low level routines. Use other names for the device level functions that you write.

2) Use the low level function add_device() to add your device to the device_table. The device table is a statically defined array which supports n devices, where n is defined by the macro _NDEVICE found in stdio.h. The structure representing a device is also defined in stdio.h and is composed of the following fields:

name	string for device name
flags	specifies whether device supports multiple streams or not
function pointers	pointers to the device level functions: ❑ CLOSE ❑ LSEEK ❑ OPEN ❑ READ ❑ RENAME ❑ WRITE ❑ UNLINK

The first entry in the device table is pre-defined to be the host device on which the debugger is running. The low level routine add_device() finds the first empty position in the device table and initializes the device fields with the passed in arguments. For a complete description of the add_device function, see page CG:5-30

3) Once the device is added call fopen() to open a stream and associate it with that device. Use *devicename***:***filename* as the first argument to fopen().

The following program illustrates adding and using a device for C I/O:

```
#include <stdio.h>

/*************************************************************************/
/* Declarations of the user-defined device drivers                       */
/*************************************************************************/
extern int my_open(char *path, unsigned flags, int fno);
extern int my_close(int fno);
extern int my_read(int fno, char *buffer, unsigned count);
extern int my_write(int fno, char *buffer, unsigned count);
extern int my_lseek(int fno, long offset, int origin);
extern int my_unlink(char *path);
extern int my_rename(char *old_name, char *new_name);

main()
{
   FILE *fid;

   add_device("mydevice", _MSA, my_open, my_close, my_read, my_write, my_lseek,
                                my_unlink, my_rename);

   fid = fopen("mydevice:test","w");

   fprintf(fid,"Hello, world\n");

   fclose(fid);
}
```

5.3 Header Files

Each runtime-support function is declared in a *header file*. Each header file declares the following:

- A set of related functions (or macros)
- Any types that you need to use the functions
- Any macros that you need to use the functions

These are the header files that declare the runtime-support functions:

```
assert.h       limits.h       setjmp.h       string.h
ctype.h        math.h         stdef.h        time.h
errno.h        mvp.h          stdio.h
float.h        stdarg.h       stdlib.h
```

In order to use a runtime-support function, you must first use the #include preprocessor directive to include the header file that declares the function. For example, the isdigit function is declared by the ctype.h header. Before you can use the isdigit function, you must first include ctype.h:

```
#include <ctype.h>
   .
   .
   .
   val = isdigit(num);
```

You can include headers in any order. You must include a header before you reference any of the functions or objects that it declares.

Subsections 5.3.1 through 5.3.12 describe the header files that are included with the TMS320C8x C compiler. Section 5.4, *Summary of Runtime-Support Functions and Macros*, lists the functions that these headers declare.

5.3.1 Diagnostic Messages (assert.h)

The assert.h header defines the assert macro, which inserts diagnostic failure messages into programs at runtime. The assert macro tests a runtime expression. If the expression is true (nonzero), the program continues running. If the expression is false, the macro outputs a message that contains the expression, the source filename, and the line number of the statement that contains the expression; then the program terminates (via the abort function).

The assert.h header refers to another macro named NDEBUG (assert.h does not define NDEBUG). If you have defined NDEBUG as a macro name when you include assert.h, then assert is turned off and does nothing. If NDEBUG *is not* defined, assert is enabled.

5.3.2 Character-Typing and Conversion (ctype.h)

The ctype.h header declares functions that test (type) and convert characters.

For example, a character-typing function may test a character to determine whether it is a letter, a printing character, a hexadecimal digit, etc. These functions return a value of *true* (a nonzero value) or *false* (0).

The character-conversion functions convert characters to lower case, upper case, or ASCII, and return the converted character.

Character-typing functions have names in the form **is***xxx* (for example, *isdigit*). Character-conversion functions have names in the form **to***xxx* (for example, *toupper*).

The ctype.h header also contains macro definitions that perform these same operations; the macros run faster than the corresponding functions. The typing macros expand to a lookup operation in an array of flags (this array is defined in ctype.c). The macros have the same name as the corresponding functions, but each macro is prefixed with an underscore (for example, *_isdigit*).

5.3.3 Error Reporting (errno.h)

Errors can occur in a math function if invalid parameter values are passed to the function or if the function returns a result that is outside the defined range for the type of the result. When this happens, a variable named errno is set to the value of one of the following macros:

- **EDOM**, for domain errors (invalid parameter)
- **ERANGE**, for range errors (invalid result)

C code that calls a math function can read the value of errno to check for error conditions. The errno variable is declared in errno.h and defined in errno.c.

5.3.4 Limits (float.h and limits.h)

The float.h and limits.h headers define macros that expand to useful limits and parameters of the TMS320C8x's numeric representations. Table 5–1 and Table 5–2 list these macros and the limits they are associated with.

Table 5–1. Macros That Supply Integer-Type Range Limits (limits.h)

Macro	Value	Description
CHAR_BIT	8	Number of bits in type char
SCHAR_MIN	–128	Minimum value for a signed char
SCHAR_MAX	127	Maximum value for a signed char
UCHAR_MAX	255	Maximum value for an unsigned char
CHAR_MIN	SCHAR_MIN	Minimum value for a char
CHAR_MAX	SCHAR_MAX	Maximum value for a char
SHRT_MIN	–32768	Minimum value for a short int
SHRT_MAX	32767	Maximum value for a short int
USHRT_MAX	65535	Maximum value for an unsigned short int
INT_MIN	–2147483648	Minimum value for an int
INT_MAX	2147483647	Maximum value for an int
UINT_MAX	4294967295	Maximum value for an unsigned int
LONG_MIN	–2147483648	Minimum value for a long int
LONG_MAX	2147483647	Maximum value for a long int
ULONG_MAX	4294967295	Maximum value for an unsigned long int

Note: Negative values in this table are defined as expressions in the actual header file so that their type is correct.

Table 5–2. Macros That Supply Floating-Point Range Limits (float.h)

Macro	MP Value	PP Value	Description
FLT_RADIX	2	2	Base or radix of exponent representation
FLT_ROUNDS	1	1	Rounding mode for floating-point addition
FLT_DIG DBL_DIG LDBL_DIG	6 15 15	6 6 6	Number of decimal digits of precision for a float, double, or long double
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	24 53 53	24 24 24	Number of base-FLT_RADIX digits in the mantissa of a float, double, or long double
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	–125 –1021 –1021	–125 –125 –125	Minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized float, double, or long double
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	128 1024 1024	128 128 128	Maximum negative integer such that FLT_RADIX raised to that power minus 1 is a representable finite float, double, or long double
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	1.19209290e–07 2.22044605e–16 2.22044605e–16	1.19209290e–07 1.19209290e–07 1.19209290e–07	Minimum positive float, double, or long double number x such that $1.0 + x \neq 1.0$
FLT_MIN DBL_MIN LDBL_MIN	1.17549435e–38 2.22507386e–308 2.22507386e–308	1.17549435e–38 1.17549435e–38 1.17549435e–38	Minimum positive float, double, or long double
FLT_MAX DBL_MAX LDBL_MAX	3.40282347e+38 1.79769313e+308 1.79769313e+308	3.40282347e+38 3.40282347e+38 3.40282347e+38	Maximum float, double, or long double
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	–37 –307 –307	–37 –37 –37	Minimum negative integers such that 10 raised to that power is in the range of normalized floats, doubles, or long doubles
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	38 308 308	38 38 38	Maximum positive integers such that 10 raised to that power is in the range of representable finite floats, doubles, or long doubles

Note: The precision of some of the values in this table has been reduced for readability. Refer to the float.h header file supplied with the compiler for the full precision carried by the processor.

Key to prefixes:

FLT_ applies to type float.
DBL_ applies to type double.
LDBL_ applies to type long double.

5.3.5 Floating-Point Math (math.h)

The math.h header defines several trigonometric, exponential, and hyperbolic math functions. These math functions expect arguments of type double and return values of type double. Except where indicated, all trigonometric functions use angles expressed in radians.

The math.h header also defines one macro named HUGE_VAL; the math functions use this macro to represent out-of-range values. When a function produces a floating-point return value that is too large to be represented, it returns HUGE_VAL instead.

5.3.6 MVP Specific Functions (mvp.h)

The mvp.h header declares several functions, macros, and data structures that allow you greater access to MVP specific features from C. The header file includes:

- NOCACHE macros that you can use to bypass the cache when making memory reference on the MP. For more information on when and how to use these macros, see subsection 4.2.3, *Cache Bypass Loads/Stores in C*.
- Function declarations for accessing six MP instructions from C: clean, command, disable, flush, lmo, and rmo.
- Definitions for MP and PP control registers. Subsections 3.3.2 and 4.4.2 explain how to access control registers from C.
- Macros that provide an easy mechanism for accessing MP memory-mapped transfer controller registers and video controller frame registers from C.
- A declaration for the PP function memtrans() which implements a memory copy (similar to memcpy()) using the PP's packet transfer.

For more information on each of the functions and macros, see Section 5.5, *Runtime-Support Functions.*

5.3.7 Nonlocal Jumps (setjmp.h)

The setjmp.h header defines one type, one macro, and one function for bypassing the normal function call and return discipline. These include:

- jmp_buf, an array type suitable for holding the information needed to restore a calling environment

- setjmp, a macro that saves its calling environment in its jmp_buf argument for later use by the longjmp function
- longjmp, a function that uses its jmp_buf argument to restore the program environment

5.3.8 Variable Arguments (stdarg.h)

Some functions can have a variable number of arguments whose types can differ; such functions are called *variable-argument functions*. The stdarg.h header declares three macros and a type that help you to use variable-argument functions:

- The three macros are *va_start*, *va_arg*, and *va_end*. These macros are used when the number and type of arguments can vary each time a function is called.
- The type *va_list* is a pointer type that can hold information for va_start, va_end, and va_arg.

A variable-argument function can use the macros declared by stdarg.h to step through its argument list at runtime, when it knows the number and types of arguments actually passed to it. You must ensure that a call to a variable argument function has visibility to a prototype for the variable-argument function in order for the arguments to be handled correctly.

5.3.9 Standard Definitions (stddef.h)

The stddef.h header defines two types and two macros. The types include:

- *ptrdiff_t*, a signed integer type that is the data type resulting from the subtraction of two pointers; *and*
- *size_t*, an unsigned integer type that is the data type of the *sizeof* operator.

The macros include:

- The *NULL* macro, which expands to a null pointer constant(0), *and*
- The *offsetof(type, identifier)* macro, which expands to an integer that has type size_t. The result is the value of an offset in bytes to a structure member (*identifier*) from the beginning of its structure (*type*).

These types and macros are used by several of the runtime-support functions.

5.3.10 Input/Output Functions (stdio.h)

The stdio.h header defines seven macros, two types, and a number of functions. The types include:

- *size_t*, an unsigned integer type that is the data type of the *sizeof* operator. The original declaration is in stddef.h.
- *fpos_t*, an unsigned long type that can uniquely specify every position within a file.
- *FILE*, a structure to record all the information necessary to control a stream.

The macros include:

- The *NULL* macro, which expands to a null pointer constant(0). Originally declared in stddef.h. It will not be redefined if it has already been defined.
- *BUFSIZ*, a macro which expands to the size of the buffer that setbuf() uses.
- *EOF*, the end-of-file marker.
- *FOPEN_MAX*, a macro that expands to the largest number of files that can be open at one time.
- *FILENAME_MAX*, a macro that expands to the length of the longest file name in characters.
- *L_tmpnam*, a macro that expands to the longest filename string that tmpnam() can generate.
- *TMP_MAX*, a macro that expands to the maximum number of unique filenames that tmpnam() can generate.

The functions are listed in Table 5–3.

5.3.11 General Utilities (stdlib.h)

The stdlib.h header declares several functions, one macro, and two types. The macro is named RAND_MAX, and it returns the largest value returned by the rand() function. The types include:

- *div_t*, a structure type that is the type of the value returned by the div function, *and*
- *ldiv_t*, a structure type that is the type of the value returned by the ldiv function.

The functions include:

- String conversion functions that convert strings to numeric representations
- Searching and sorting functions that allow you to search and sort arrays
- Sequence-generation functions that allow you to generate a pseudo-random sequence and choose a starting point for a sequence
- Program-exit functions that allow your program to terminate normally or abnormally, *and*
- Integer-arithmetic that is not provided as a standard part of the C language.

5.3.12 String Functions (string.h)

The string.h header declares standard functions that allow you to perform the following tasks with character arrays (strings):

- Move or copy entire strings or portions of strings,
- Concatenate strings,
- Compare strings,
- Search strings for characters or other strings, *and*
- Find the length of a string.

In C, all character strings are terminated with a 0 (null) character. The string functions named **str***xxx* all operate according to this convention. Additional functions that are also declared in string.h allow you to perform corresponding operations on arbitrary sequences of bytes (data objects), where a 0 value does not terminate the object. These functions have names such as **mem***xxx*.

When you use functions that move or copy strings, be sure that the destination is large enough to contain the result.

5.3.13 Time Functions (time.h)

The time header declares one macro, several types, and functions that manipulate dates and times. Times are represented in two ways:

- As an arithmetic value of type **time_t**. When expressed in this way, a time is represented as a number of seconds since 12:00 AM January 1, 1900. The time_t is a synonym for the type unsigned long.
- As a structure of type **struct_tm**. This structure contains members for expressing time as a combination of years, months, days, hours, minutes, and seconds. A time represented like this is called broken-down time. The structure has the following members.

```
int   tm_sec;     /* seconds after the minute (0–59) */
int   tm_min;     /* minutes after the hour (0–59)   */
int   tm_hour;    /* hours after midnight (0–23)     */
int   tm_mday;    /* day of the month (1–31)         */
int   tm_mon;     /* months since January (0–11)     */
int   tm_year;    /* years since 1900 (0–99)         */
int   tm_wday;    /* days since Saturday (0–6)       */
int   tm_yday;    /* days since January 1 (0–365)    */
int   tm_isdst;   /* daylight savings time flag      */
```

A time, whether represented as a time_t or a struct_tm, can be expressed from different points of reference.

- Calendar time represents the current Gregorian date and time.
- Local time is the calendar time expressed for a specific time zone.

Local time may be adjusted for daylight savings time. Obviously, local time depends on the time zone. The time.h header declares a structure type called tmzone and a variable of this type called _tz. You can change the time zone by modifying this structure, either at runtime or by editing tmzone.c and changing the initialization. The default time zone is central standard time, USA.

The basis for all the functions in time.h are two system functions: clock and time. Time provides the current time (in time_t format), and clock provides the system time (in arbitrary units). The value returned by clock can be divided by the macro CLOCKS_PER_SEC to convert it to seconds. Since these functions and the CLOCKS_PER_SEC macro are system specific, only stubs are provided in the library. To use the other time functions, you must supply custom versions of these functions.

5.4 Summary of Runtime-Support Functions and Macros

Table 5–3 contains a table showing all of the runtime-support header files (in alphabetical order) provided with the TMS320C8x ANSI C compilers. Most of the functions described in the summary table are per the ANSI standard and behave exactly as described in the standard. The header file mvp.h, however, contains C functions specific to the TMS320C8x architecture. You will find complete descriptions of these functions in Section 5.5, *Runtime-Support Functions*.

Table 5–3. Summary of Runtime Support Functions and Macros

Error Messages (assert.h)	**Description**
void **assert**(int expression); ‡	Inserts diagnostic messages into programs
Character Typing and Conversion (ctype.h)	**Description**
int **isalnum(**char c); †	Tests c to see if it's an alphanumeric-ASCII character.
int **isalpha**(char c); †	Tests c to see if it's an alphabetic-ASCII character.
int **isascii**(char c); †	Tests c to see if it's an ASCII character.
int **iscntrl**(char c); †	Tests c to see if it's a control character.
int **isdigit**(char c); †	Tests c to see if it's a numeric character.
int **isgraph**(char c); †	Tests c to see if it's any printing character except a space.
int **islower**(char c); †	Tests c to see if it's a lowercase alphabetic ASCII character.
int **isprint**(char c); †	Tests c to see if it's a printable ASCII character (including a space).
int **ispunct**(char c); †	Tests c to see if it's an ASCII punctuation character.
int **isspace**(char c); †	Tests c to see if it's an ASCII spacebar, tab (horizontal or vertical), carriage return, formfeed, or newline character.
int **isupper**(char c); †	Tests c to see if it's an uppercase ASCII alphabetic character.
int **isxdigit**(char c); †	Tests c to see if it's a hexadecimal digit.
char **toascii**(char c); †	Masks c into a legal ASCII value.
char **tolower**(int char c); †	Converts c to lowercase if it's uppercase.
char **toupper**(int char c); †	Converts c to uppercase if it's lowercase.

† Expands inline if –x is used
‡ Macro
§ Expands inline unless –x0 is used

Floating-Point Math (math.h)	Description
double **acos**(double x);	Returns the arc cosine of x.
double **asin**(double x);	Returns the arc sine of x.
double **atan**(double x);	Returns the arc tangent of x.
double **atan2**(double y, double x);	Returns the arc tangent of y/x.
double **ceil**(double x); †	Returns the smallest integer greater equal to x.
double **cos**(double x);	Returns the cosine of x.
double **cosh**(double x);	Returns the hyperbolic cosine of x.
double **exp**(double x);	Returns the exponential function of x.
double **fabs**(double x); §	Returns the absolute value of x.
double **floor**(double x); †	Returns the largest integer less than or equal to x.
double **fmod**(double x, double y); †	Returns the floating-point remainder of x/y.
double **frexp**(double value, int *exp);	Breaks value into a normalized fraction and an integer power of 2.
double **ldexp**(double x, int exp);	Multiplies x by an integer power of 2.
double **log**(double x);	Returns the natural logarithm of x.
double **log10**(double x);	Returns the base-10 logarithm of x.
double **modf**(double value, int *iptr);	Breaks value into a signed integer and a signed fraction.
double **pow**(double x, double y);	Returns x raised to the power y.
double **sin**(double x);	Returns the sine of x.
double **sinh**(double x);	Returns the hyperbolic sine of x.
double **sqrt**(double x);	Returns the nonnegative square root of x.
double **tan**(double x);	Returns the tangent of x.
double **tanh**(double x);	Returns the hyperbolic tangent of x.
MVP Specific (mvp.h)	**Description**
void **clean**(void *buf, long len);	Cleans the buffer in cache. (MP only)
void **command**(long val);	Sends a command word to MVP processors. (MP only)
long **disable**(void);	Disables the MVP interrupts. (MP only)
void **flush**(void *buf, long len);	Flushes the buffer from cache. (MP only)
char ***_gen_addr**(far char *p);	Returns p if p points to external memory, otherwise the PP number is returned.
long **lmo**(long val);	Returns the bit number of the leftmost 1. (MP only)
char***memtrans** (char *to, char *from, unsigned int length);	Similar to memcpy, but uses the transfer controller (PP only)
long **rmo**(long val);	Returns the bit number of the rightmost 1. (MP only)

† Expands inline if –x is used
‡ Macro
§ Expands inline unless –x0 is used

Nonlocal Jumps (setjmp.h)	Description
int **setjmp**(jmp_buf env); ‡	Saves calling environment for use by longjmp.
void **longjmp**(jmp_buf env, int returnval);	Uses jmp_buf argument to restore a previously saved environment.

Variable-Argument Functions (stdarg.h)	Description
type **va_arg**(va_list ap, *type*); ‡	Accesses the next argument of type *type* in a variable-argument list.
void **va_end**(va_list ap); ‡	Resets the calling mechanism after using va_arg.
void **va_start**(va_list ap, *lastarg*); ‡	Initializes ap to point to the first operand in the variable-argument list.

C I/O (stdio.h)	Description
int **add_device** (char *name, unsigned flags, int (*dopen)(), int (*dclose)(), int (*dread)(), int (*dwrite)(), long (*dlseek)(), int (*dunlink)(), int (*drename)());	
void **clearerr**(FILE *p);	Clears the EOF and error indicators for the stream that p points to.
int **fclose**(FILE *iop);	Flushes the stream that iop points to and closes the file associated with that stream.
int **feof**(FILE *p);	Tests the EOF indicator for the stream that p points to.
int **ferror**(FILE *p);	Tests the error indicator for the stream that p points to.
int **fflush**(register FILE *iop);	Flushes the I/O buffer for the stream that iop points to.
int **fgetc**(register FILE *fp);	Reads the next character in the stream that fp points to.
int **fgetpos**(FILE *iop, fpos_t *pos);	Stores the object that pos points to to the current value of the file position indicator for the stream that iop points to.
char ***fgets**(char *ptr, register int size, register FILE *iop);	Reads the next size –1 characters from the stream that iop points to into array ptr.
FILE ***fopen**(const char *file, const char *mode);	Opens the file that file points to; mode points to a string describing how to open the file.
int **fprintf**(FILE *iop, const char *format, ...);	Writes to the stream that iop points to.
int **fputc**(int c, register FILE *fp);	Writes a single character, c, to the stream that fp points to.

† Expands inline if –x is used
‡ Macro
§ Expands inline unless –x0 is used

C I/O (stdio.h) (continued)	**Description**
int **fputs**(const char *ptr, register FILE *iop);	Writes the string that ptr points to to the stream that iop points to.
size_t **fread**(void *ptr, size_t size, size_t count, FILE *iop);	Reads from the stream pointed to by iop and stores the input to the memory area pointed to by ptr.
FILE ***freopen**(const char *file, const char *mode, register FILE *iop);	Opens the file that file points to using the stream that iop points to; mode points to a string describing how to open the file.
int **fscanf**(FILE *iop, const char *fmt, ...);	Reads from the stream pointed to by iop.
int **fseek**(register FILE *iop, long offset, int ptrname);	Sets the file position indicator for the stream pointed to by iop.
int **fsetpos**(FILE *iop, const fpos_t *pos);	Sets the file position indicator for the stream that iop points to pos. The pointer pos must be a value from fgetpos() on the same stream.
long **ftell**(FILE *iop);	Obtains the current value of the file position indicator for the stream that iop points to.
size_t **fwrite**(const void *ptr, size_t size, size_t count, register FILE *iop);	Writes a block of data from the memory pointed to by ptr to the stream that iop points to.
int **getc**(FILE *p);	A macro that calls fgetc()
int **getchar**(void);	A macro that calls fgetc() and supplies stdin as the argument.
char ***gets**(char *ptr);	Performs the same function as fgets() using stdin as the input stream.
void **perror**(const char *s);	Maps the error number in s to a string and prints the error message.
int **printf**(const char *format, ...);	Performs the same function as fprintf but uses stdout as its output stream.
int **putc**(int x, FILE *p);	A macro that performs like fputc().
int **putchar**(int x);	A macro that calls fput() and uses stdout as the output stream.
int **puts**(const char *ptr);	Writes the string pointed to by ptr to stdout.
int **remove** (const char *file);	
int **rename** (const char *old, const char *new);	
void **rewind** (register FILE *iop);	
void **setbuf** (register FILE *iop, char *buf);	
int **setvbuf** (register FILE *iop, register char *buf, register int type, register size_t size);	
int **scanf** (const char *fmt, ...);	
int **sprintf** (char *string, const char *format, ...);	

† Expands inline if –x is used
‡ Macro
§ Expands inline unless –x0 is used

int **sscanf** (const char *str, const char *fmt, ...);	
FILE ***tmpfile**(void);	Creates a temporary file.
char ***tmpnam**(char *s);	Generates a string that is a valid filename.
int **vfprintf** (FILE *iop, const char *format, va_list ap);	
int **vprintf** const char *format, va_list ap);	
int **vsprintf** (char *string, const char *format, va_list ap);	

General Utilities (stdlib.h)	**Description**
int **abs**(int val); §	Returns the absolute value of val.
void **abort**(void)	Terminates a program abnormally.
int **atexit**(void (*func)(void));	Registers the function pointed to by func, called without arguments at program termination.
double **atof**(const char *st); †	Converts a string to a floating-point value.
int **atoi**(char *st);	Converts a string to a floating-point value.
long **atol**(char *st); †	Converts a string to a long integer.
void *__bsearch__(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *,const void *));	Searches through an array of n objects for the object that key points to.
void ***calloc** (size_t num, size_t size);	Allocates and clears memory for num objects, each of size bytes.
div_t **div**(int numer, int denom);	Divides numer by denom producing a quotient and a remainder.
void **exit**(int status);	Terminates a program normally.
void **free**(void *ptr);	Deallocates memory space allocated by malloc, calloc, or realloc.
long **labs**(long val); §	Returns the absolute value of val.
ldiv_t **ldiv** (long numer, long denom);	Divides numer by denom.
int **ltoa**(long n, char *buffer);	Converts n to the equivalent string.
void ***malloc**(size_t size);	Allocates memory for an object of size bytes.
void ***memalign**(size_t aln, size_t size);	Allocates memory for an object of size bytes alaigned to an aln byte boundary.
void **minit**(void);	Resets all the memory previously allocated by malloc, calloc, or realloc.
void **qsort**(void *base, size_t nmemb, size_t size, int (*compar) (void));	Sorts an array of nmemb members; base points to the first member of the unsorted array, and size specifies the size of each member.
int **rand**(void);	Returns a sequence of pseudorandom integers in the range 0 to RAND_MAX.

† Expands inline if –x is used
‡ Macro
§ Expands inline unless –x0 is used

void ***realloc**(void *ptr, size_t size);	Changes the size of memory pointed to by ptr to size.
void **srand**(unsigned seed);	Resets the random number generator.
double **strtod**(char *st, char **endptr);	Converts a string to a floating-point value.
long **strtol**(char *st, char **endptr, int base);	Converts a string to a long integer.
unsigned long **strtoul(**char *st, char **endptr**,** int base**);**	Converts a string to an unsigned long integer.

String Functions (string.h)	**Description**
void *__memchr__(const void *cs, int c, size_t n); †	Finds the first occurrence of c in the first n characters of cs.
int __memcmp__(const void *cs, const void *ct, size_t n); †	Compares the first n characters of cs to ct.
void *__memcpy__(void *s1, const void *s2, size_t n);	Copies n characters from s1 to s2.
void *__memmove__(void *s1, const void *s2, size_t n);	Moves n characters from s1 to s2.
void *__memset__(void *mem, int ch, size_t n); †	Copies the value of ch into the first n characters of mem.
char *__strcat__(char *string1, const char *string2);	Appends string2 to the end of string1.
char *__strchr__(const char *string, int c);	Finds the first occurrence of character c in string.
int __strcmp__(const char *string1, const char *string2);	Compares strings and returns one of the following values: <0 if string1 is less than string2; =0 if string1 is equal to string2; >0 if string1 is greater than string2.
int __strcoll__(const char *string1, const char *string2);	Compares strings and returns one of the following values, depending on the locale: <0 if string1 is less than string2; =0 if string1 is equal to string2; >0 if string1 is greater than string2.
char *__strcpy__(char *dest, const char *src);	Copies string src into dest.
size_t __strcspn__(const char *string, const char *chs);	Returns the length of the initial segment of string that is made up entirely of characters that are not in chs.
char *__strerror__(int errno);	Maps the error number in errno to an error message string.
size_t __strlen__(const char *string);	Returns the length of a string.
char *__strncat__(char *to, const char *from, size_t n); †	Appends up to n characters from from to to.
int __strncmp__(const char *string1, const char *string2, size_t n); †	Compares up to n characters in two strings.
char *__strncpy__(char *to, const char *from, size_t n); †	Copies up to n characters of from to to.

† Expands inline if –x is used
‡ Macro
§ Expands inline unless –x0 is used

char *__strpbrk__(const char *string, const char *chs);	Locates the first occurrence in string of *any* character from chs.
char *__strrchr__(const char *string, int c); †	Finds the last occurrence of character c in s.
size_t __strspn__(const char *string, const char *chs);	Returns the length of the initial segment of string, which is entirely made up of characters from chs.
char *__strstr__(const char *string1, const char *string2);	Finds the first occurrence of string2 in string1.
char *__strtok__(char *str1, const char *str2);	Breaks str1 into a series of tokens, each delimited by a character from str2.
size_t __strxfrm__(char *to, const char *from, size_t n);	Transforms n characters from from, to to.

Time Functions (time.h)	**Description**
char ***asctime**(const struct tm *timeptr);	Converts a time to a string.
clock_t **clock**(void);	Determines the processor time used.
char ***ctime**(const struct time *timeptr); †	Converts time to a string.
double **difftime**(time_t time1, time_t time0);	Returns the difference between two calendar times.
struct tm ***gmtime**(const time_t *timer);	Converts local time to Greenwich Mean Time.
struct tm ***localtime**(const time_t *timer);	Converts time_t value to broken down time.
time_t **mktime**(struct tm *tptr);	Converts broken down time to a time_t value.
size_t **strftime**(char *out, size_t maxsize, const char *format, const struct tm *timeptr);	Formats a time into a character string.
time_t **time**(time_t *timer);	Returns the current calendar time.

† Expands inline if –x is used
‡ Macro
§ Expands inline unless –x0 is used

5.5 Runtime-Support Functions

This section contains complete descriptions of all non-ANSI run-time-support functions provided with the TMS320C8x compilers.

add_device Add device to device table

Syntax

```
#include <stdio.h>
int add_device(char *name,
               unsigned flags,
               int (*dopen)(),
               int (*dclose)(),
               int (*dread)(),
               int (*dwrite)(),
               int (*dlseek)(),
               int (*dunlink)(),
               int (*drename)());
```

Defined in

lowlev.obj in xx_cio[l].lib

Description

The **add_device** function adds a device record to the device table allowing that device to be used for input/ output from C. The first entry in the device table is pre-defined to be the host device on which the debugger is running. The function add_device() finds the first empty position in the device table and initializes the fields of the structure that represents a device.

To open a stream on a newly-added device use fopen() with a string of the format *devicename*:*filename* as the first argument.

Parameters

name	character string denoting the device name
flag	Device characteristics. Currently there two defined flags that can be used as parameters to add_device() **_SSA** denotes that device supports only one open stream at a time **_MSA** denotes that device supports multiple open streams More can be added by defining them in stdio.h
dopen, dclose, dread, dwrite, dlseek, dunlink, drename	Function pointers to the device drivers are called by the low level functions to perform I/O on the specified device. You must define these functions with the interface specified in subsection 5.2.1. The device drivers for the host that the TMS320C8x debugger is run on are included in the C I/O library.

Return Value If successful, add_device() returns 0, otherwise –1.

Example This example will add the device *mydevice* to the device table, open a file named *test* on that device and associate it with the file **fid*, print the string *Hello, world* into the file, and close the file.

```
#include <stdio.h>

/*************************************************************************/
/* Declarations of the user-defined device drivers                       */
/*************************************************************************/
extern int my_open(char *path, unsigned flags, int fno);
extern int my_close(int fno);
extern int my_read(int fno, char *buffer, unsigned count);
extern int my_write(int fno, char *buffer, unsigned count);
extern int my_lseek(int fno, long offset, int origin);
extern int my_unlink(char *path);
extern int my_rename(char *old_name, char *new_name);

main()
{

   FILE *fid;
   add_device(”mydevice”, _MSA, my_open, my_close, my_read, my_write, my_lseek,
                                my_unlink, my_rename);

   fid = fopen(”mydevice:test”,”w”);

   fprintf(fid,”Hello, world\n”);

   fclose(fid);
}
```

clean — Clean buffer in cache

Syntax

```
#include <mvp.h>
void clean (void *buf, long len);
```

Defined in

clean.asm in mp_rts.src

Description

The clean function cleans a buffer from cache. This function allows you to control the MP's data cache from a C program. Argument buf is a pointer to a buffer area in memory (which may span one or more 64-byte subblocks) and len is the buffer's length in bytes. The function uses the dcachec instruction to clean each cache subblock that contains a portion of the buffer.

The procedure for cleaning a subblock in the data cache is:

1) If the subblock's dirty flag is set, write the contents of the cache subblock back to main memory.

2) Reset the subblock's dirty flag.

If the buffer occupies any portion of a subblock, the entire subblock is cleaned. If len is less than 0, the function returns without cleaning any portion of the cache. For more information about the MP's data cache, see Section 6.2, *Data Cache,* of the *MVP Master Processor User's Guide*. For more information about the MP dcache instruction, see page MP:10-74 in the *MVP Master Processor User's Guide*.

Example

In the following example, the clean function cleans two buffers in cache. Since the buf1 is aligned to a 64-byte boundary, the buffer spans a single subblock, and only that subblock is cleaned. The second buffer, buf2, is 65 bytes long, and, thus, definitely spans two subblocks, so both subblocks are cleaned. If buf1 was not explicitly aligned, the first clean instruction would clean one or two subblocks, depending on how the buffer happened to be aligned.

```
#include <mvp.h>
#pragma DATA_ALIGN (buf1, 64);

char buf1[64];
char buf2[65];

clean(buf1, sizeof(buf1));
clean(buf2, sizeof(buf2));
```

command

Send command word

Syntax

#include <mvp.h>
void command (long *val***);**

Defined in

command.asm in mp_rts.src

Description

The command function sends a command word to the various processors on the MVP device by executing an MP cmnd instruction. The argument val is the 32-bit command word for the cmnd instruction. For more information about the MP cmnd instruction see page MP:10-70 of the *MVP Master Processor User's Guide*.

Example

The following example sends several hardware commands to various processors:

```
#include <mvp.h>

#define UNHALT (1<<30)
int n;
command(0x400000FF);  /* Halt all parallel processors */

n=3;
command(UNHALT | (1<<n&7)); /* Unhalt PP #3          */

command(0x00002100);  /* MP sends self interrupt     */
```

disable — Disable interrupts

Syntax

```
#include <mvp.h>
long disable(void);
```

Defined in

disable.asm in mp_rts.src

Description

The disable function disables interrupts and returns the original value of the MVP master processor's 32-bit IE (interrupt enable) register. The function loads the value 0x1 into the IE register. This disables all interrupts individually, but leaves the global interrupt enable flag set to 1 so that integer multiplies and other operations performed by the floating-point unit can be performed while interrupts are disabled.

The function uses the swcr instruction to simultaneously read the original contents of IE and to write a new value of 0x1 to IE. This avoids the timing hazard that would occur if separate rdcr and wrcr instructions were used to read IE and write to IE. If an interrupt were to occur between the read and write operations and if the IE register was modified before the return from interrupt, the value of IE that was originally read would no longer be valid. The disable function ensures that this type of failure cannot occur.

Example

The following example shows how to make sure that a section of critical code is not interrupted before it has completed.

```
#include <mvp.h>      /* MVP hardware functions   */
long save_ie;
save_ie = disable();  /* disable interrupts       */

< Insert critical code section here. >

IE = save_ie;         /* restore interrupts       */
```

flush

Flush buffer from cache

Syntax

#include <mvp.h>
void flush(void **buf*, **long** *len*);

Defined in

flush.asm in mp_rts.src

Description

The flush function flushes a buffer from cache. This function allows you to control the MVP master processor's data cache from a C program. Argument buf is a pointer to a buffer area in memory, and argument len is the buffer's length in bytes. The buffer may span one or more cache-aligned 64-byte subblocks in memory. The function uses the dcachef instruction to flush each cache subblock that contains a portion of the buffer.

The procedure for flushing a subblock from the data cache is as follows:

- If the subblock's dirty flag is set, write the contents of the cache subblock back to main memory.
- Reset both the dirty flag and the present flag for the subblock.

If the buffer occupies only a portion of any subblock, the entire subblock is still flushed. If argument len is 0, the function returns without flushing any portion of the cache. For more information about the MP's data cache, see Section 6.2, *Data Cache*, of the *MVP Master Processor User's Guide*. For more information about the MP dcache instruction, see page MP:10-74 in the *MVP Master Processor User's Guide*.

Example

In the following example, the flush function flushes a 64-byte buffer from cache. If the buffer happens to be aligned to an even 64-byte subblock boundary in memory, the buffer spans a single subblock, and only this subblock is flushed. Otherwise, the buffer straddles the boundary between two subblocks, and both subblocks are flushed.

```
#include <mvp.h>
char buf[64];
flush(buf, sizeof(buf));
```

lmo — Leftmost one

Syntax

#include <mvp.h>
long lmo(long *val***);**

Defined in

lmo.asm in mp_rts.src

Description

The lmo function returns the bit number of the leftmost 1 in the argument. Argument val is treated as a 32-bit value whose bits are numbered from 0 to 31, where bit 0 is the LSB (the rightmost bit position) and bit 31 is the MSB (the leftmost bit position).

For nonzero arguments, the return value is in the range 0 to 31. If the argument is 0, the function returns a value of –1. For more information on the MP's lmo instruction, see page MP:10-119 in the *MVP Master Processor User's Guide*.

Example

```
#include <mvp.h>    /* MVP hardware functions */

long i, j;

i = 0x80000000;
j = lmo(i);         /* return value j = 31 */

i = 0x00000001;
j = lmo(i);         /* return value j = 0 */

i = 0;
j = lmo(i)          /* return value j = –1 */

i = 0x00C59AF8;
j = lmo(i)          /* return value j = 23 */

i = 0x7200F100;
j = lmo(i)          /* return value j = 30 */

i = –1;
j = lmo(i)          /* return value j = 31 */
```

memalign

Align heap

Syntax

#include <stdlib.h>
void *memalign(size_t *_aln*, **size_t** *_size*);

Defined in

memalign.asm in mp_rts.src

Description

The memalign function performs exactly like the ANSI standard malloc function, except that it returns a pointer to a block of memory that is aligned to an *aln* byte boundary. Thus if *size* is 128, and *aln* is 16, memalign will return a pointer to a 128 byte block of memory aligned on a 16 byte boundary.

memtrans

Memory copy using packet transfer

Syntax

#include <mvp.h>
**char *memtrans (register char *****to*, **register char *****from*,
register unsigned int *length*);

Defined in

memtrans.c in pp_rts.src

Description

The memtrans function performs the same operation as the memcpy() function. Instead of doing a byte by byte copy as memcpy() does, however, memtrans() sets up and issues a packet transfer request to the transfer controller. For copying data between external and on-chip memory, a packet transfer is more efficient. For more information on using packet transfers, see Chapter 12, *Packet Transfers*, in the *MVP Parallel Processor User's Guide*.

rmo — Rightmost one

Syntax

#include <mvp.h>
long rmo(long *val***);**

Defined in

rmo.asm in mp_rts.src

Description

The rmo function returns the bit number of the rightmost 1 in the argument. Argument val is treated as a 32-bit value whose bits are numbered from 0 to 31, where bit 0 is the LSB (the rightmost bit position) and bit 31 is the MSB (the leftmost bit position).

For nonzero arguments, the return value is in the range 0 to 31. If the argument is 0, the function returns a value of –1.

Example

```
#include <mvp.h>    /* MVP hardware functions */

long i, j;

i = 0x80000000;
j = rmo(i);         /* return value j = 31 */

i = 0x00000001;
j = rmo(i);         /* return value j = 0 */

i = 0;
j = rmo(i)          /* return value j = –1 */

i = 0x00C59AF8;
j = rmo(i)          /* return value j = 3 */

i = 0x7200F100;
j = rmo(i)          /* return value j = 8 */

i = –1;
j = rmo(i)          /* return value j = 0 */
```

Chapter 6

Library Build Utility

When using the TMS320C8x C compilers, you can compile your code under a number of configurations and options, that are not necessarily compatible. Since it would be cumbersome to include all possible combinations in individual runtime-support libraries, this package includes the source files, mp_rts.src and pp_rts.src, which contain all runtime-support functions not related to I/O, and all floating-point support functions. By using the mpmk and ppmk utilities described in this chapter you can custom-build your runtime-support libraries for the options you select.

Topics

6.1 Invoking the Library Build Utility

The general syntax for invoking the library build utility is:

mpmk [*–options*] *src_arch1* [**–l** *obj.lib1*] [*src_arch2* [**–l** *obj.lib2*]]
or...
ppmk [*–options*] *src_arch1* [**–l** *obj.lib1*] [*src_arch2* [**–l** *obj.lib2*]]

mpmk is the command that invokes the MP library build utility.

ppmk is the command that invokes the PP library build utility.

options affect how the source files are extracted and compiled. (Options are discussed in Section 6.2, *Options Summary* and below.)

src_arch is the name of a source archive file. For each source archive named, mpmk or ppmk builds an object library according to the runtime model specified by the command line options.

–l*obj.lib* is the optional object library name. If you do not specify a name for the library, mpmk or ppmk will use the name of the source archive and append a **.lib** suffix. Only one library name is allowed for each archive file; extras are ignored. If you name an object library, the object library name must be supplied immediately after the source archive name from which it is derived.

The mpmk or ppmk utility runs the shell program mpcl or ppcl on each source file in the archive to either compile or assemble it. It then collects all the object files into the output library. All the tools must be in your PATH. *The utility ignores and disables the environment variables* TMP, C_OPTION, *and* C_DIR.

6.2 Options Summary

Most of the options that are included on the command line correspond directly to options of the same name used with the compiler and assembler. However, some of the options are specific to the library build utility

Library build utility specific options

The following options apply only to the library build utility.

––c C source files contained in the source archive are extracted from the library and left in the current directory after the utility has completed execution.

––h instructs mpmk or ppmk to use header files contained in the source archive and leave them in the current directory after the utility has completed execution.

––k instructs the mpmk or ppmk utility to overwrite files. By default, the utility will abort when it attempts to extract a source file or create a library if a file of the same name already exists.

––u instructs mpmk or ppmk not to use the header files contained in the source archive when building the object library. If the desired headers are already in the current directory, there is no reason to reinstall them. This option also gives you some flexibility in modifying runtime-support functions to suit your application.

––v prints progress information to the screen during execution of the utility.

––q runs the utility in quiet mode. Only errors will be printed to the screen.

The following commands build the standard runtime support libraries as object libraries named mp_rts.lib and pp_rts.lib. The libraries are optimized with inline function expansion (–x and –o). The example assumes that the runtime support headers already exist in the current directory (––u).

```
mpmk ––u –o –x mp_rts.src –1 mp_rts.lib

ppmk ––u –o –x pp_rts.src –1 pp_rts.lib
```

Other options

The other options that can be used with the mpmk or ppmk utilities correspond directly to the options that the compiler uses. These options are described in detail in subsection 1.1.3, *Compiler Options*. The following table provides a summary of the options that you can use with the utilities. The options are the same for mpmk and ppmk, except –mc (PP only).

General Options	**Effect**
–g	symbolic debugging
–v2	set support for silicon version 2 (default)
–v3	set support for silicon version 3
Parser Options	**Effect**
–pk	K&R compatible
–pw	suppress warning messages
–p?	enable trigraph expansion
Optimizer Options	**Effect**
–o0	level 0 register optimization
–o1	level 1 +local optimization
–o2 (or –o)	level 2 +global optimization
–o3	level 3 +file level optimization
Inlining Options	**Effect**
–x1	default inlining level
–x2 (or –x)	defines _INLINE and invokes optimizer at level 2
Runtime Model Options	**Effect**
–ma	assumes aliased variables
–mc	place string constants in external memory (PP only)
–me	use little endian ordering
Type Checking Options	**Effect**
–tf	relax prototype checking
–tp	relax pointer combination checking
Assembler Options	**Effect**
–as	keep labels as symbols
File Specifiers	**Effect**
–ea	set default extension for assembly files
–eo	set default extension for object files

Chapter 7

Assembler Description

The MVP PP and MP assemblers translate assembly language source code into machine language object files in common object file format (COFF). This chapter discusses how to invoke the assembler, the various options available to you, and the format of the output files.

Chapter 11, *Assembler Error Messages*, describes the error messages produced by the assembler.

Topics

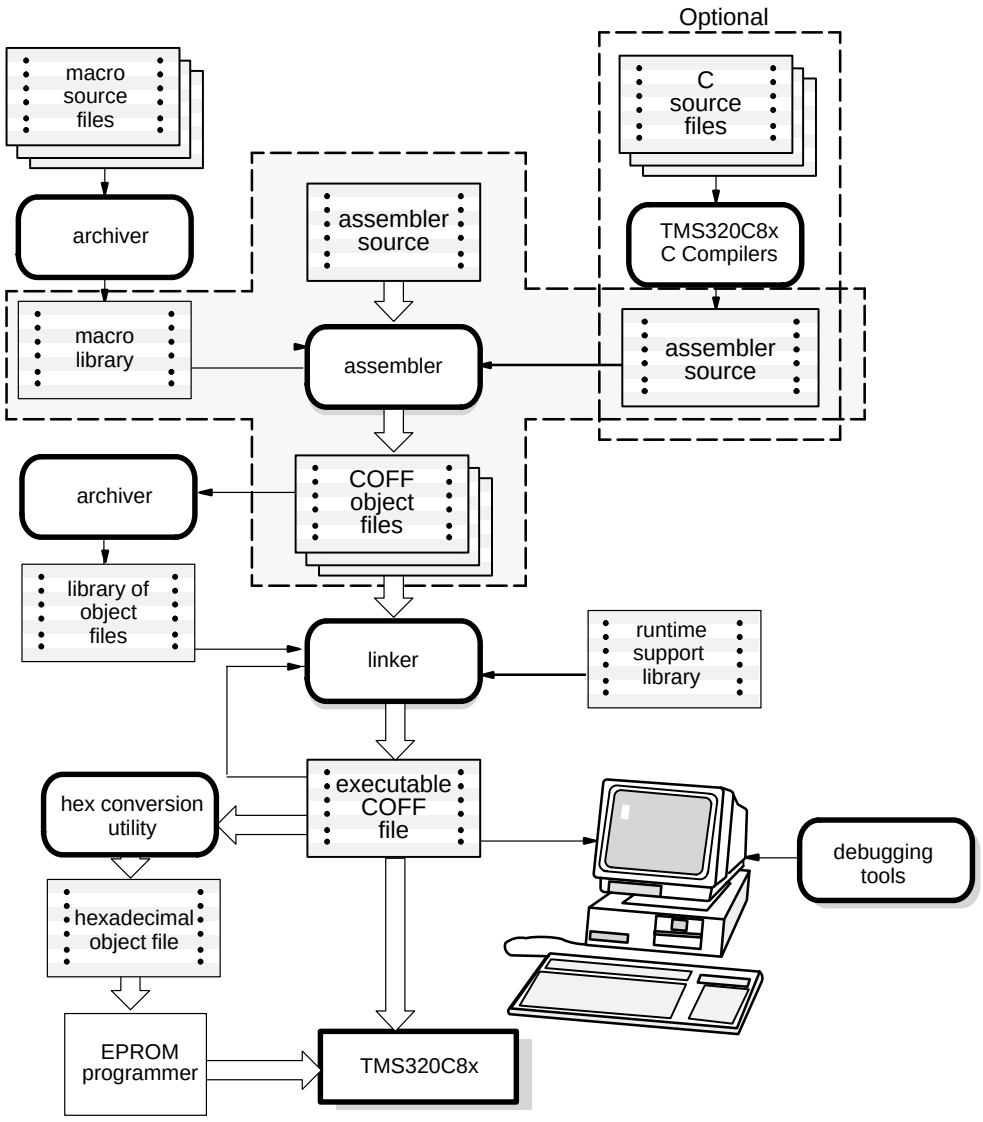
7.1 Assembler Development Flow

The assemblers perform the following tasks:

- Process the source statements in a text file to produce a relocatable object file
- Produce a source listing (if requested) and provide you with control over this listing
- Allow you to segment your code into sections and maintain an SPC (section program counter) for each section of object code
- Define and reference global symbols and append a cross-reference listing to the source listing (if requested)
- Assemble conditional blocks
- Support macros, allowing you to define macros inline or in a library
- Assemble TMS320C8x code

Figure 7–1 illustrates the assembler's role in the assembly language development flow. The assembler accepts assembly language source files as input. The TMS320C8x assemblers also accept assembly language files created by the TMS320C8x C compilers.

Figure 7–1. Assembler Development Flow



7.2 Invoking the Assembler

To invoke the assembler, enter the following:

ppasm	[*input file*	[*object file*	[*listing file*]]]	[**–***options*]
mpasm	[*input file*	[*object file*	[*listing file*]]]	[**–***options*]

ppasm is the command that invokes the PP assembler.

mpasm is the command that invokes the MP assembler.

input file names the assembly language source file. If you do not supply an extension, the assembler assumes that the input file has the default extension .asm for the MP and .s for the PP. If you do not supply an input filename when you invoke the assembler, the assembler will prompt you for one.

object file names the object file that the assembler creates. If you do not supply an extension, the MP assembler uses .obj as a default extension and the PP assembler uses .o. If you do not supply an object file, the assembler creates a file that uses the input filename with the .obj (MP) or .o (PP) rm extension.

listing file names the optional listing file that the assembler can create. If you do not supply a name for a listing file, the assembler does not create one unless you use the –l option. In this case, the assembler uses the input filename. If you do not supply an extension, the assembler uses .lst as a default extension.

options identifies the assembler options that you want to use. Options are not case sensitive and can appear anywhere on the command line, following the command. Precede each option with a dash (–).You can string the options together; for example, **–lc** is equivalent to **–l –c.** The valid assembler options are as follows:

–c makes case significant. For example, the symbols ABC and abc are not equivalent. If you do not use this option, case is insignificant.

–d defines predefined symbols. Up to 64 symbols can be defined from the command line. The format of the –d option is **–d** *name* [**=***def*]. The *name* is the name of the constant and *def* is the value assigned to the constant. If **=***def* is omitted, *name* will be set to one.

–e produces little endian COFF. Do not link little endian code with big endian code. The default is big endian.

–i specifies a directory where the assembler can find files named by the .copy, .include, or .mlib directives. The format of the –i option is **–i***pathname*. You can specify up to 64 directories in this manner; each pathname must be preceded by the –i option.

–l (lowercase L) produces a listing file.

–q (quiet) suppresses the banner and all progress information.

–s puts all defined symbols in the object file's symbol table. Usually, the assembler puts only global symbols into the symbol table. When you use –s, symbols that are defined as labels or as assembly-time constants are also placed in the symbol table.

–v*n* set silicon version.

- **–v2** sets support for pre-production silicon version 2(default).
- **–v3** sets support for production silicon version 3.

–x produces a cross-reference table and appends it to the end of the listing file. The listing file is produced when you use –x regardless of whether you used the –l (lowercase L) option or not.

7.3 Naming Alternate Directories for Assembler Input

The .copy, .include, and .mlib directives tell the assembler to use code from external files. The .copy and .include directives tell the assembler to read source statements from another file, and the .mlib directive names a library that contains macro functions. Chapter 9, *Assembler Directives*, contains examples of the .copy, .include, and .mlib directives. The syntax for these directives is:

```
.copy   ”filename”
.include ”filename”
.mlib    ”filename”
```

The filename names either a copy/include file from which the assembler reads statements or a macro library that contains macro definitions. The filename may be a complete pathname or a filename with no path information. If you provide a pathname, the assembler uses that path and does not look for the file in any other directories. If you do not provide path information, the assembler searches directories for the file in the following order:

1) The directory that contains the current source file. The current source file is the file being assembled when the .copy, .include, or .mlib directive is encountered.

2) Any directories named with the –i assembler option.

3) Any directories set with the environment variable A_DIR.

You can augment the assembler’s directory search algorithm by using the –i assembler option or the environment variable.

7.3.1 –i Assembler Option

The –i assembler option names an alternate directory that contains copy/include files or macro libraries. The format of the –i option is as follows:

mpasm **–i***pathname* *source filename*
ppasm **–i***pathname* *source filename*

You can use up to 64 –i options per invocation; each –i option names one pathname. In assembly source, you can use the .copy, .include, or .mlib directive without specifying any path information. If the assembler doesn't find the file in the directory that contains the current source file, it searches the paths provided by the –i options.

For example, assume that a file called source.asm is in the current directory; source.asm contains the following directive statement:

```
.copy "copy.asm"
```

Pathname for copy.asm	Invocation Command
`/mvp/files/copy.asm`	`mpasm –i/mvp/files source.asm`

The assembler first searches for copy.asm in the current directory because source.asm is in the current directory. Then the assembler searches in the directory named with the –i option.

7.3.2 Environment Variable (A_DIR)

An environment variable is a system symbol that you define and assign a string to. The assembler uses the environment variable **A_DIR** to name alternate directories that contain copy/include files or macro libraries. The command for assigning the environment variable is as follows:

```
setenv     A_DIR "pathname;another pathname ... "
```

The pathnames are directories that contain copy/include files or macro libraries. You can separate the pathnames with a semicolon or with blanks. In assembly source, you can use the .copy, .include, or .mlib directive without specifying any path information. If the assembler doesn't find the file in the directory that contains the current source file or in directories named by –i, it searches the paths named by the environment variable.

For example, assume that a file called source.asm contains these statements:

```
.copy "copy1.asm"
.copy "copy2.asm"
```

Pathname	Invocation Command
/mvp/files/copy1.asm /dsys/copy2.asm	setenv A_DIR=/dsys; ppasm –i/mvp/files source.asm

The assembler first searches for copy1.asm and copy2.asm in the current directory because source.asm is in the current directory. Then the assembler searches in the directory named with the –i option and finds copy1.asm. Finally, the assembler searches the directory named with A_DIR and finds copy2.asm.

Note that the environment variable remains set until you reboot the system or reset the variable by entering this command:

```
unsetenv A_DIR
```

7.4 Source Listings

A source listing shows source statements and the object code they produce. To obtain a listing file, invoke the assembler with the –l (lowercase L) option.

At the top of each source listing page are two banner lines, a blank line, and a title line. Any title supplied by a .title directive is printed on this line; a page number is printed to the right of the title. If you don't use the .title directive, the title area is left blank. The assembler inserts a blank line below the title line.

Each line in the source file may produce a line in the listing file that shows a source statement number, an SPC value, the object code assembled, and the source statement. A source statement may produce more than one word of object code. The assembler lists the SPC value and object code on a separate line for each additional word. Each additional line is listed immediately following the source statement line.

Field 1: source statement number

Line Number

The source statement number is a decimal number. The assembler numbers source lines as it encounters them in the source file; some statements increment the line counter but are not listed (for example, .title statements and statements following a .nolist are not listed). The difference between two consecutive source line numbers indicates the number of statements in the source file that are not listed.

Include File Letter

The assembler may precede a line with a letter; the letter indicates that the line is assembled from an include file.

Nesting Level Number

The assembler may precede a line with a number; the number indicates the nesting level of macro expansions and loop blocks.

Field 2: section program counter

This field contains the section program counter, or SPC, value (hexadecimal). Each section (.text, .data, .bss, and named sections) maintains a separate SPC. Some directives do not affect the SPC; they leave this field blank.

Field 3: object code

This field contains the hexadecimal representation of the object code. All machine instructions and directives use this field to list object code. This field also indicates the relocation type by appending one of the following characters to the end of the field:

!	undefined external reference
’	.text relocatable
”	.data relocatable
+	.sect relocatable
–	.bss/.usect relocatable

Field 4: source statement field

This field contains the characters of the source statement as they were scanned by the assembler. The assembler accepts a maximum line length of 200 characters. Spacing in this field is determined by the spacing in the source statement.

Example 7–1 shows an example of an assembler listing with each of the four fields identified.

Example 7–1.PP Assembler Source Listing

```
       1                                    .global Unsigned_Half
       2                                    .global div_aux_tab
       3
       4          0000000000000021  Div:    .set    d1
       5          0000000000000022  Num:    .set    d2
       6          0000000000000023  Tmp:    .set    d3
       7          0000000000000023  Quot:   .set    d3
       8          0000000000000022  Rem:    .set    d2
       9 00000000                   Unsigned_Half:
      10
      11                                    .eval   0, x
      12                            COEF:   .loop
      13                                    .eval   x + 1, x
      14                                    .break  x == 3
      15                                    .endloop
1                                           .eval   0 + 1, x
1                                           .break  1 == 3
1                                           .eval   1 + 1, x
1                                           .break  2 == 3
1                                           .eval   2 + 1, x
1                                           .break  3 == 3
      16 00000000 978118C000000000!         a9  = div_aux_tab
      17 00000008 BB89280000141C50          Div = 0 – Div << 16
      18                                        ||  mf = a15
      19 00000010 FA83508A201F474C          Tmp = Div +   Num
      20                                        ||  br = [z] *(a9)
      21 00000018 7413408C205F474C          Tmp = divi(Div, Num=Num)
      22                                        ||  br = [v] *(a9 + [1])
      23 00000020 7413608E80776B4C          Tmp = divi(Div, Num=Tmp [n] Num)
      24                                        ||  lrse2 = *(a9 – [1])
      25 00000028 7413608E80104100          Tmp = divi(Div, Num=Tmp [n] Num)
      26 00000030 7413608E80104100          Tmp = divi(Div, Num=Tmp [n] Num)
      27
      28                                    .copy ”inc4.asm”
 A     1                            * Macro for incrementing
 A     2
 A     3                            inc4    .macro  reg1, reg2, reg3, reg4
 A     4                                    reg1 = reg1 + 1
 A     5                                    reg2 = reg2 + 1
 A     6                                    reg3 = reg3 + 1
 A     7                                    reg4 = reg4 + 1
 A     8                                    .endm
      29
      30 00000038 7413608E80104100  LoopUH: Tmp = divi(Div, Num=Tmp [n] Num)
      31 00000040 8B22600E40104100          Num = [n] Tmp
      32 00000048 8B23400000104140          Quot= mf
      33 00000050 BB6A480000104100          Rem = Num >> 16
      34 00000058                           inc4 d1, d2, d3, d4
1        00000058 8A21208000104100          d1 = d1 + 1
1        00000060 8A22408000104100          d2 = d2 + 1
1        00000068 8A23608000104100          d3 = d3 + 1
1        00000070 8A24808000104100          d4 = d4 + 1
```

Field 1 Field 2 Field 3 Field 4

7.5 Cross-Reference Listings

A cross-reference listing shows symbols and their definitions. To obtain a cross-reference listing, invoke the assembler with the –x option or use the .option directive. The assembler will append the cross-reference to the end of the source listing.

Example 7–2. Assembler Cross-Reference Listing

```
LABEL                              VALUE       DEFN    REF

.MVP_MP_BIG                     00000000'          0
.MVP_MP_LITTLE                  00000000'          0
.MVP_PP_BIG                     00000001'          0
.MVP_PP_LITTLE                  00000000'          0
.TMS320MVP_MP                   00000000'          0
.TMS320MVP_PP                   00000001'          0
.mvp_mp_big                     00000000'          0
.mvp_mp_little                  00000000'          0
.mvp_pp_big                     00000001'          0
.mvp_pp_little                  00000000'          0
.tms320mvp_mp                   00000000'          0
.tms320mvp_pp                   00000001'          0
COEF                            00000000'         12
LoopUH                          00000038'         30
Unsigned_Half                   00000000'          9      1
div_aux_tab                         REF                   2     16
```

label column contains each symbol that was defined or referenced during the assembly.

value column contains a 8-digit hexadecimal number, which is the value assigned to the symbol *or* a name that describes the symbol's attributes. A value may also be followed by a character that describes the symbol's attributes. Table 7–1 lists these characters and names.

definition (DEFN) column contains the statement number that defines the symbol. This column is blank for undefined symbols.

reference (REF) column lists the line numbers of statements that reference the symbol. A blank in this column indicates that the symbol was never used.

Table 7–1. Symbol Attributes

Character or Name	Meaning
REF	External reference (global symbol)
UNDF	Undefined
‘	Symbol defined in a .text section
”	Symbol defined in a .data section
+	Symbol defined in a .sect section
–	Symbol defined in a .bss or .usect section

Chapter 8

Assembly Language Source Files

The assembler translates assembly language source files into machine language object files in common object file format (COFF). There are two separate assemblers for the MVP, and two separate assembly languages, but the format of the assembly language files accepted by both assemblers is similar. Source files can contain the following assembly language elements:

- MVP PP assembly language instructions (see the *MVP Parallel Processor User's Guide*, Chapter 8, *The PP Assembly Language Instruction Set*) **or**
- MVP MP assembly language instructions (see the *MVP Master Processor User's Guide*, Chapter 10, *The MP Assembly Language Instruction Set*)
- Assembly language directives (see *Assembler Directives*, Chapter 9)
- Macro directives (see *Macro Language*, Chapter 10)

This chapter tells you how to create assembly language source files using these elements. Additionally, it explains the syntax of the assembly language source statement and the syntax and format of symbols, constants, characters, and expressions.

Topics

8.1 Source Statement Format

MVP assembly language source programs consist of source statements that can include assembler directives, assembly language instructions or operations, macro directives, and comments. Source statement lines can be as long as the source file format allows, but the assembler reads a maximum of 200 characters per line. If a statement contains more than 200 characters, the assembler truncates the line and issues a warning.

The next several lines show examples of MP source statements:

```
SYM1      .set      2      ; Symbol SYM1 = 2
Begin:    add r6, r7, r8
          .word     016h
```

An MP source statement can contain four ordered fields. The general syntax for MP source statements is as follows:

[*label*][*:*] **mnemonic** [*operand list*] [*;comment*]

The next several lines show examples of PP source statements:

```
begin:
   d0 = [le] d7 * d0
   d2 =u d6 * d5
      || d4 = ealu (d3, d2\\d0, %d0)
```

A PP source statement can contain three ordered fields. The general syntax for PP source statements is as follows:

[*label*][*:*] **operation** [*;comment*]

Follow these guidelines:

- All statements must begin with a label, a blank, an asterisk, or a semicolon.
- Labels are optional; if used, they must begin in column 1.
- One or more blanks must separate each field. Tab characters are equivalent to blanks.
- Comments are optional. Comments that begin in column 1 can begin with an asterisk or a semicolon (* or ;), but comments that begin in any other column must begin with a semicolon.

Label field

Labels are optional for all assembly language instructions and for most (but not all) assembler directives. When used, a label must begin in column 1 of a source statement. A label can contain up to 32 alphanumeric characters (A–Z, a–z, 0–9, _, and $). Labels are case sensitive, and the first character cannot be a number. A label can be followed by a colon (:); the colon is not treated as part of the label name. If you don't use a label, the first character position must contain a blank, a tab, a semicolon, or an asterisk.

When you use a label, its value is the current value of the section program counter (the label points to the statement it's associated with). If, for example, you used the .word directive to initialize several words, a label would point to the first word. In the following example, the label Begin has the value 0h.

```
4 00000000 000000000000000a  Begin:  .word 0Ah, 3, 7
  00000004 0000000000000003
  00000008 0000000000000007
```

A label on a line by itself is a valid statement. It assigns the current value of the section program counter to the label; this is equivalent to the following directive statement:

```
label .set  $  ;  $ provides the current value of the SPC
```

When a label appears on a line by itself, it points to the location of the instruction on the next line (the SPC is not incremented):

```
5 00000010                    Begin:
6 00000010 0000000000000003           .word 3
```

Labels that appear on a line alone are always aligned to a valid instruction boundary (8 bytes on the PP or 4 bytes on the MP).

MP mnemonic field

The MP mnemonic field follows the label field. The mnemonic field must not start in column 1, or it will be interpreted as a label. The mnemonic field can contain one of the following opcodes:

- An MP machine-instruction mnemonic (such as rdcr, rmo, jsr), see the *MVP Master Processor User's Guide*, Chapter 10, *The MP Assembly Language Instruction Set*.
- An assembler directive (such as .data, .list, .set), see Chapter 9, *Assembler Directives*.
- A macro directive (such as .macro, .var, .mexit), see Chapter 10, *Macro Language*.
- A macro call, see Chapter 10, *Macro Language*.

MP operand list field

The MP operand field is a list of operands that follow the mnemonic field. An operand can be:

- A constant (see Section 8.2, *Constants*),
- A symbol (see Section 8.4, *Symbols*), *or*
- A combination of constants and symbols in an expression (see Section 8.5, *Expressions*).

You must separate operands with commas. MP operands can occur in any of the valid MP addressing modes. See The *MVP Master Processor User's Guide,* Section 10.12, *Alphabetical Instruction Reference*.

PP operation field

The PP operation field can start anywhere following the optional label. The operation field must not start in column 1, or it will be interpreted as a label. The operation field consists of:

- An assembler directive (see Chapter 9, *Assembler Directives*) *or*,
- One or more parallel operations specified within a PP assembly language instruction.

As described in the *MVP Parallel Processor User's Guide,* Chapter 8, *The PP Assembly Language Instruction Set*, PP instructions often specify parallel operations performed by the multiplier, ALU, global address unit, and/or the local address unit.

Any of these individual operations can be omitted in an instruction. The assembler will insert the appropriate bits in the instruction to result in no operation by the corresponding hardware. The *MVP Parallel Processor User's Guide,* Section 8.10, *Parallel-Operation Combinations* discusses in detail the supported combinations of parallel operations.

The first operation specified within a PP instruction requires simply the algebraic assembly expression for the operation. Operations that are to performed in parallel are joined by **||**. The parallel operations that make up a PP instruction can either be placed in the source code on one line:

```
d7 = d6 * d5 ||d4 = d3 + d2 ||*a8 = d1 ||d0 = *a0 ; Comments
```

or on separate lines:

```
d7 = d6 * d5       ; Multiply comment.
   ||d4 = d3 + d2  ; ALU comment.
   ||*a8 = d1      ; Global transfer comment.
   ||d0 = *a0      ; Local transfer comment.
```

Throughout this user's guide, individual operations within an instruction are placed on separate lines as shown above. This enhances code readability and allows each operation to have a separate comment field. Also as a matter of the coding style, parallel operations (those following ||) are indented in the source code to make it easier to identify operations forming a single PP instruction.

Comment field

A comment can begin in any column and can extend to the end of the source line. A comment can contain any ASCII character, including blanks. Comments are printed in the assembly source listing, but they do not affect the assembly.

A source statement that contains only a comment is valid. If it begins in column 1, it can start with a semicolon (;) or an asterisk (*). Comments that begin anywhere else must begin with a semicolon. The asterisk identifies a comment only in column 1.

8.2 Constants

Both the MP and PP assemblers support seven types of constants:

- Binary integer constants
- Octal integer constants
- Decimal integer constants
- Hexadecimal integer constants
- Character constants
- Floating point constants
- Assembly-time constants

The assembler maintains each integral constant internally as a 32-bit quantity. Note that constants *are not sign-extended.* For example, the constant 0FFH is equal to 00FF (base 16) or 255 (base 10); it *does not* equal –1.

Binary integers

A binary integer constant is a string of up to 32 binary digits (0s and 1s) followed by the suffix B (or b). If fewer than 32 digits are specified, the assembler right-justifies the value and zero-fills the unspecified bits. These are examples of valid binary constants:

00000000B	Constant equal to 0_{10} or 0_{16}
0100000b	Constant equal to 32_{10} or 20_{16}
01b	Constant equal to 1_{10} or 1_{16}
11111000B	Constant equal to 248_{10} or $0F8_{16}$

Octal integers

An octal integer constant is a string of up to 10 octal digits (0 through 7) followed by the suffix Q (or q). These are examples of valid octal constants:

10Q	Constant equal to 8_{10} or 8_{16}
100000Q	Constant equal to 32768_{10} or 8000_{16}
226q	Constant equal to 150_{10} or 96_{16}

Decimal integers

A decimal integer constant is a string of decimal digits, ranging from –2,147,483,647 to 4,294,967,295. These are examples of valid decimal constants:

1000	Constant equal to 1000_{10} or $3E8_{16}$
–32768	Constant equal to -32768_{10} or 8000_{16}
25	Constant equal to 25_{10} or 19_{16}

Hexadecimal integers

A hexadecimal integer constant is a string of up to eight hexadecimal digits preceded by an 0x or followed by the suffix H (or h). Hexadecimal digits include the decimal values 0–9 and the letters A–F and a–f. A hexadecimal constant using the h suffix notation must begin with a decimal value (0–9). If fewer than eight hexadecimal digits are specified, the assembler right-justifies the bits. These are examples of valid hexadecimal constants:

78h	Constant equal to 120_{10} or 0078_{16}
0fh	Constant equal to 15_{10} or $000F_{16}$
37ACH	Constant equal to 14252_{10} or $37AC_{16}$
0xABCD	Constant equal to 43981_{10} or $ABCD_{16}$

Character constants

A character constant is a string of one or two characters enclosed in single quotes. The characters are represented internally as 8-bit ASCII characters. Two consecutive single quotes are required to represent each single quote that is part of a character constant. A character constant consisting only of two single quotes is valid and is assigned the value 0. If only one character is specified, the assembler right-justifies the bits. These are examples of valid character constants:

'ab'	Represented internally as 00006162_{16}
'C'	Represented internally as 00000043_{16}
'''D'	Represented internally as 00004427_{16}

(three single quotes, a D, and another single quote)

Note the difference between character constants and character strings (see Section 8.3, *Character Strings*). A character constant represents a single integer value; a string is a list of characters.

Assembly-time constants

If you use the .set directive to assign a value to a symbol, the symbol becomes a constant. To use this constant in expressions, the value assigned to it must be absolute. For example:

```
shift3    .set   3
          srl    shift3, 32, r7, r8
```

You can also use the .set directive to assign symbolic constants for register names. In this case, the symbol becomes a synonym for the register:

```
source1   .set   r7
source2   .set   r8
dest      .set   r9

   add source1, source2, dest
```

8.3 Character Strings

A character string is a string of characters enclosed in *double* quotes. Double quotes that are part of character strings are represented by two consecutive double quotes. The maximum length of a string varies and is defined for each directive that requires a character string. Characters are represented internally as 8-bit ASCII characters.

These are examples of valid character strings:

”sample program”	defines the 14-character string, *sample program*
”PLAN ””C”””	defines the 8-character string, *PLAN ”C”*

Character strings are used for the following:

- ❑ Filenames, as in .copy ”filename”
- ❑ Section names, as in .sect ”section name”
- ❑ Operands of .string, .byte, .word, or .half directives

8.4 Symbols

Symbols are used as labels, constants, and substitution symbols. A symbol name is a string of up to 32 alphanumeric characters (A–Z, a–z, 0–9, $, and _). The first character in a symbol cannot be a number, and symbols cannot contain embedded blanks. The symbols you define are case sensitive; for example, the assembler recognizes ABC, Abc, and abc as three unique symbols. You can override case sensitivity with the –c assembler option. Symbols are valid only during the assembly in which they are defined, unless you use the .global directive to declare them as external.

Labels

Symbols used as labels become symbolic addresses associated with locations in the program. Labels used locally within a file must be unique. Mnemonic opcodes and assembler directive names (without the dot prefix) are valid label names.

Labels can also be used as the operands of .global, .ref, .def, or .bss directives; for example:

```
          .global   label1

label2    nop
          add    label2, r1, r2
          br.a   label1
```

Constants

Symbols can be set to constant values. By using constants, you can equate meaningful names with constant values. The .set and .struct/.tag/.endstruct directives enable you to set constants to symbolic names. Symbolic constants *cannot* be redefined. The following example shows how these directives can be used:

```
K      .set   1024            ;  constant definitions
maxbuf.set   2*K

item   .struct                ;  item structure definition
       .int   value           ;  constant offsets value = 0
       .int   delta           ;  constant offsets delta = 1
i_len  .endstruct

array  .tag   item            ;  array declaration
       .bss   array, i_len*K

       add array.delta, r7, array.delta
                              ; array.delta += r7
```

The assembler also has several predefined symbolic constants, which are discussed in the next section.

Symbolic constants

The assemblers have several predefined symbols, including the following:

- **$**, **the dollar sign character**, represents the current value of the relocatable section program counter (SPC)
- **Register symbols**
  - The MP assembler recognizes symbols for the MP registers, such as r0–r31, and a0–a3. For a complete description of MP registers, see the *MVP Master Processor User's Guide,* Chapter 2, *Master Processor Registers*.
  - The PP assembler recognizes symbols for the PP registers, such as pc, ipa, d0–d7, and tag0–tag3. For a complete description of PP registers, see the *MVP Parallel Processor User's Guide,* Chapter 7, *Summary of PP Registers*.
- **Address mode symbols**, such as :s and :m used for specific addressing modes on the MP.

Substitution symbols

Symbols can be assigned a string value (variable). This enables you to alias character strings by equating them to symbolic names. Symbols that represent character strings are called substitution symbols. When the assembler encounters a substitution symbol, its string value is substituted for the symbol name. Unlike symbolic constants, substitution symbols can be redefined.

A string can be assigned to a substitution symbol anywhere within a program; for example:

```
.asg  "zero",  r0    ;  MP zero register

add   zero, r7, r8
```

When you are using macros, substitution symbols are important because macro parameters are actually substitution symbols that are assigned a macro argument. The following code shows how substitution symbols are used in macros:

```
add3  .macro src1, src2, src3, dest  ;  add3 macro


      add   src1, src2, dest
      add   dest, src3, dest
      .endm

      .global loc1, loc2, loc3, loc4
*add3 invocation
      add3  r1, r2, r3, r4
```

For more information about macros, see Chapter 10, *Macro Language*.

Local labels

Local labels are special labels whose scope and effect are temporary. A local label has the form **$***n*, where n is a decimal digit in the range 0–9. For example, $4 and $1 are valid local labels.

Normal labels must be unique (they can be declared only once), and they can be used as constants in the operand field. Local labels, however, can be undefined and defined again.

A local label can be undefined, or reset, in one of four ways:

- By using the .newblock directive
- By changing sections (using a .sect, .text, .ptext, or .data directive)
- By entering an include file (specified by the .include or .copy directive)
- By leaving an include file (specified by the .include or .copy directive)

Example 8–1 shows code that declares and uses a local label legally. Example 8–2 shows code that declares and uses a local label illegally.

Example 8–1. Legal Local Label Usage

```
Label1:   d6 = *(dba + 0xffc)
          d7 =ub *dba, a8 = dba + 0xfff
$1        *(a2 = pba + 0x200) = d7
$2        *(pba + 0xfc) = a2
          *(a1=dba+0x800) = x1
          .newblock
          d1 = *(a11=dba + 0x8000)
$1        d6 = *(dba + 0xffc)
          a2 = &*(a3 = pba + 0x1b8)
          d7 =ub *dba, a8 = dba + 0xfff
```

Example 8–2. Illegal Local Label Usage

```
Label1:   d6 = *(dba + 0xffc)
          d7 =ub *dba, a8 = dba + 0xfff
$1        *(a2 = pba + 0x200) = d7
$2        *(pba + 0xfc) = a2
          *(a1=dba+0x800) = x1
          d1 = *(a11=dba + 0x8000)
$1        d6 = *(dba + 0xffc)
          ; WRONG $1 is multiply defined
          a2 = &*(a3 = pba + 0x1b8)
          d7 =ub *dba, a8 = dba + 0xfff
```

Local labels are especially useful in macros. If a macro contains a normal label and is called more than once, the assembler issues a multiple-definition error. If you use a local label and .newblock within a macro, however, the local label is used and reset each time the macro is expanded.

Up to ten local labels can be in effect at one time. After you undefine a local label, you can define it and use it again. Local labels do not appear in the object code symbol table.

8.5 Expressions

An expression is a constant, a symbol, or a series of constants and symbols separated by arithmetic operators. The range of valid expression values is a 32-bit range, –2147483648 to 4294967295. Three main factors influence the order of expression evaluation:

Parentheses

Expressions enclosed in parentheses are always evaluated first.

8/(4/2) = 4, but 8/4/2=1

You cannot substitute braces ([ ]) or brackets ({ }) for parentheses.

Precedence groups

Operators, listed in Table 8–1 and Table 8–2, are divided into precedence groups. When parentheses do not determine the order of expression evaluation, the highest precedence operation is evaluated first.

8 + 4/2 = 10 (4/2 is evaluated first)

Left-to-right evaluation

When parentheses and precedence groups do not determine the order of expression evaluation, the expressions are evaluated from left to right, except for unary operators which are evaluated from right to left.

8/4*2 = 4, but 8/(4*2) = 1

8.5.1 Operators

Table 8–1 and Table 8–2 list the operators that can be used in expressions according to precedence group. Within groups with the same precedence, evaluation is left-to-right, except unary operators for the MP, which are evaluated right-to-left.

Table 8–1. Operators Used in MP Expressions (Precedence)

Group 1 (Highest Precedence) Right-to-Left Evaluation		Group 2	
+	Unary plus (positive expression)	*	Multiplication
–	Unary minus (negative expression)	/	Division
~	1s complement	%	Modulo
!	Logical not	<<	Left -shift
		>>	Right-shift
Group 3		**Group 4 (Lowest Precedence)**	
+	Addition	<	Less than
–	Subtraction	>	Greater than
^	Bitwise exclusive-OR	<=	Less than or equal to
|	Bitwise OR	>=	Greater than to equal to
&	Bitwise AND	=	(==) Equal to
||	Logical OR	!=	Not equal to
&&	Logical AND		

Notes: 1) Operators within parentheses **( )** indicate an alternate form.

2) Bitwise operations on floating-point constants will cause an error.

Table 8–2. Operators Used in PP Expressions (Precedence)

Group 1 (Highest Precedence)			
*src1***[***n***]***src1–1*	Conditional source selection†		
Group 2			
( )	Subexpression delimiters		
Group 3			
~	Bitwise NOT†	**%!**	Right-shift mask generation†
@	Multiple flags expand†	**%%!**	Non-multiple right-shift mask generation†
%	Mask generation†	**+**	Unary Plus
%%	Non-multiple mask generation†	**–**	Unary Minus
Group 4			
****	Rotate left†	**> >s**	Arithmetic (zero extended) shift right†
< <	Shift left†	**> >**	Arithmetic (zero extended) shift right†
		> >u	Logical (sign extended) shift right†
Group 5			
&	Bitwise AND		
Group 6			
^	Bitwise XOR		
Group 7			
|	Bitwise OR		
Group 8			
*****	Multiplication	**/**	Division
Group 9			
+	Addition	**–**	Subtraction
Group 10 (Lowest Precedence)			
=[*cond***]**	Conditional assignment†	**=[***cond.pro***]**	Conditional assignment with status protection†
+ =	Add to destination†	**– =**	Subtract from destination†
=	Equate†	**<**	Less than‡
>	Greater than‡	**< =**	Less than or equal to‡
> =	Greater than to equal to‡	**= =**	Equal to ‡

† These operators can only be used in instructions, not in general expressions
‡ These operators can only be used in general expressions, not in instructions

8.5.2 Expression Overflow and Underflow

The assembler checks for overflow and underflow conditions when arithmetic operations are performed at assembly time. The assembler issues a *Value Truncated* warning whenever an overflow or underflow occurs. The assembler *does not* check for overflow or underflow in multiplication.

8.5.3 Well-Defined Expressions

Some assembler directives require well-defined expressions as operands. Well-defined expressions contain only symbols or assembly-time constants that are defined before they are encountered in the expression. The evaluation of a well-defined expression must be absolute.

This is an example of a well-defined expression:

```
1000h+X
```

where X was previously defined as an absolute symbol.

8.5.4 Conditional Expressions

The assembler supports relational operators that can be used in any expression; they are especially useful for conditional assembly. Relational operators include the following:

!=	Not equal to	**==**	Equal to
<	Less than	**<=**	Less than or equal to
>	Greater than	**>=**	Greater than or equal to

These operators have the lowest precedence; each has the same precedence within the group, however, so they are evaluated from left to right. Conditional expressions evaluate to 1 if true and 0 if false.

8.5.5 Relocatable Symbols and Legal Expressions

Table 8–3 summarizes valid operations on absolute, relocatable, and external symbols. An expression cannot contain multiplication or division by a relocatable or external symbol. An expression cannot contain unresolved symbols that are relocatable with respect to other sections.

Table 8–3. Expressions With Absolute and Relocatable Symbols

If A is...	If B is...	A + B is...	A – B is...
absolute	absolute	absolute	absolute
absolute	external	external	illegal
absolute	relocatable	relocatable	illegal
relocatable	absolute	relocatable	relocatable
relocatable	relocatable	illegal	absolute †
relocatable	external	illegal	illegal
external	absolute	external	external
external	relocatable	illegal	illegal
external	external	illegal	illegal

†A and B must be in the same section; otherwise, this is illegal.

Following are examples of expressions that use relocatable and absolute symbols. These examples use four symbols that are defined in the same section:

```
          .global extern_1   ;  Defined in an external module
intern_1: .word '"D'         ;  Relocatable, defined in current module
LAB1:     .set 2             ;  LAB1 = 2
intern_2                     ;  Relocatable, defined in current module
```

❑ Example 1

The statements in this example use an absolute symbol, LAB1 (which we defined on the previous page to have a value of 2). The first statement sets sym1 to 51. The second statement sets sym2 to 27.

```
sym1   .set   LAB1   +   ((4+3)*   7)     ;   sym1 = 51
sym2   .set   LAB1   +   4   +   (3*7)         ;   sym2 = 27
```

❑ Example 2

All legal expressions can be reduced to one of two forms:

relocatable symbol ± *absolute symbol*

or

absolute value

Unary operators can be applied only to absolute values; they cannot be applied to relocatable symbols. Expressions that cannot be reduced to contain only one relocatable symbol are illegal. The first statement in the following example is valid; the statements that follow it are invalid.

```
sym3   .set   extern_1 – 10   ;Legal
sym4   .set   10–extern_1     ;Can't negate reloc. sym.
sym4   .set   –(intern_1)     ;Can't negate reloc. sym.
sym5   .set   extern_1/10     ;/ isn't an additive op.
sym6   .set   intern_1+extern_1  ;Multiple relocatables
```

Example 3

The first statement below is legal; although intern_1 and intern_2 are relocatable, their difference is absolute because they're in the same section. Subtracting one relocatable symbol from another reduces the expression to *relocatable symbol* + *absolute value*. The second statement is illegal because the sum of two relocatable symbols is not an absolute value.

```
sym7   .set   intern_1–intern_2+extern_1   ;  Legal
sym8   .set   intern_1+intern_2+extern_1   ;  Illegal
```

Example 4

An external symbol's placement in an expression is important to expression evaluation. Although the statement below is similar to the first statement in the previous example, it is illegal because of left-to-right operator precedence; the assembler attempts to add intern_1 to extern_1.

```
sym9   .set   intern_1+extern_1–intern_2   ; Illegal
```

8.6 Built-In Assembler Functions

The TMS320C8x assemblers provide several built-in functions to simplify common operations.

8.6.1 Floating-Point to Integer Conversions

The assembler provides four built-in functions which round floating-point values to integer values:

$$trunc(*expression***)**	returns the result of the *expression* rounded towards zero.
$$round(*expression***)**	returns the result of the *expression* rounded to the nearest integer.
$$floor(*expression***)**	returns the largest integer that is not greater than the *expression*.
$$ceil(*expression***)**	returns the smallest integer that is not less than the *expression*.

Each function returns a signed integer value. Here is an example that uses floating-point to integer conversions:

❑ **Example**

```
flt1  .set  1.23
flt2  .set  3.45/flt1
flt3  .set  flt2 * $$ceil (flt1)
```

This example attaches a value roughly equal to 5.61 to flt3.

8.6.2 Structure Query Functions

The assembler provides two built-in functions which help you set up and use pointers to structures in assembly language:

$$structsz(*stag***)**	returns the size of the structure associated with the structure tag *stag*.
$$structacc(*stag***)**	returns the access point of the structure associated with *stag*.

Both functions return an integral value.

Chapter 9

Assembler Directives

Assembler directives supply data and control the assembly process. The first part of this chapter (Sections 9.1 through 9.9) describes the directives according to function, and the second part (Section 9.10) is an alphabetical reference.

Topics

9.1 Directives Summary

Table 9–1 summarizes the assembler directives. (Macro directives can be found in Section 10.10, *Macro Directives Summary*.) Note that all source statements that contain a directive may have a label and a comment. To improve readability, labels and comments are not shown as part of the directive syntax.

Table 9–1. Assembler Directives Summary

Directives That Define Sections		
Mnemonic and Syntax	**Description**	**Page**
.bss *symbol*, *size in bytes* [, *alignment*]	Reserve *size* bytes in the .bss (uninitialized data) section	CG:9-26
.data	Assemble into the .data (initialized data) section	CG:9-31
.ptext	Assemble into the PP-specific .ptext section (executable code; similar to the .text section which refers only to the MP)	CG:9-76
.sect "*section name***"**	Assemble into a named (initialized) section	CG:9-64
.text	Assemble into the .text (executable code) section of the MP(.ptext refers to the PP)	CG:9-76
symbol **.usect "***section name***",** *size in bytes* [, *alignment*]	Reserve *size* bytes in a named (uninitialized) section	CG:9-83

Table 9–1. Assembler Directives Summary (Continued)

Directives That Initialize Constants (Data and Memory)		
Mnemonic and Syntax	**Description**	**Page**
.bes *size in bytes*	Reserve *size* bytes in the current section; note that a label points to the last addressable byte in the reserved space	CG:9-66
.byte $value_1$ [**,** ... **,** $value_n$]	Initialize one or more successive bytes in the current section	CG:9-28
.char $value_1$ [**,** ... **,** $value_n$]	Initialize one or more successive bytes in the current section	CG:9-28
.double *value* [**,** ... **,** $value_n$]	Initialize one or more 32-bit, IEEE single-precision, floating-point constants for PP, 64- bit, IEEE double-precision floating point constants for MP	CG:9-40
.field *value* [**,** *size in bits*]	Initialize a variable-length field	CG:9-38
.float *value* [**,** ... **,** $value_n$]	Initialize one or more 32-bit, IEEE single-precision, floating-point constants	CG:9-40
.half $value_1$ [**,** ... **,** $value_n$]	Initialize one or more16-bit half-words	CG:9-44
.int $value_1$ [**,** ... **,** $value_n$]	Initialize one or more 32-bit integers	CG:9-52
.long $value_1$ [**,** ... **,** $value_n$]	Initialize one or more 32-bit integers	CG:9-52
.short $value_1$ [**,** ... **,** $value_n$]	Initialize one or more16-bit half-words	CG:9-44
.space *size in bytes;*	Reserve *size* bytes in the current section; note that a label points to the beginning of the reserved space	CG:9-66
.string "*$string_1$***"** [**,** ... **, "***$string_n$***"**]	Initialize one or more text strings	CG:9-68
.ubyte $value_1$ [**,** ... **,** $value_n$]	Initialize one or more successive bytes in the current section	CG:9-28
.uchar $value_1$ [**,** ... **,** $value_n$]	Initialize one or more successive bytes in the current section	CG:9-28
.uhalf $value_1$ [**,** ... **,** $value_n$]	Initialize one or more 16-bit half-words	CG:9-44
.uint $value_1$ [**,** ... **,** $value_n$]	Initialize one or more 32-bit integers	CG:9-52
.ulong $value_1$ [**,** ... **,** $value_n$]	Initialize one or more 32-bit integers	CG:9-52
.ushort $value_1$ [**,** ... **,** $value_n$]	Initialize one or more 16-bit half-words	CG:9-44
.uword $value_1$ [**,** ... **,** $value_n$]	Initialize one or more 32-bit integers	CG:9-52
.word $value_1$ [**,** ... **,** $value_n$]	Initialize one or more 32-bit integers	CG:9-52

Table 9–1. Assembler Directives Summary (Continued)

Directives That Align the Section Program Counter (SPC)		
Mnemonic and Syntax	**Description**	**Page**
.align [*byte boundary*]	Align the SPC on a byte boundary specified by the parameter (must be a power of 2), or default to 4-byte boundary. Aligns to byte(1), 32-bit word (4), etc.	CG:9-23

Directives That Format the Output Listing		
Mnemonic and Syntax	**Description**	**Page**
.drlist	Enable listing of all directive lines (default)	CG:9-32
.drnolist	Inhibit listing of certain directive lines	CG:9-32
.fclist	Allow false conditional code block listing (default)	CG:9-37
.fcnolist	Inhibit false conditional code block listing	CG:9-37
.length *page length*	Set the page length of the source listing	CG:9-49
.list	Restart the source listing	CG:9-50
.mlist	Allow macro listings and loop blocks (default)	CG:9-58
.mnolist	Inhibit macro listings and loop blocks	CG:9-58
.nolist	Stop the source listing	CG:9-50
.option {*A / B / D / H / L / M / N / O / R / T / W / X*}	Select output listing options	CG:9-61
.page	Eject a page in the source listing	CG:9-63
.sslist	Allow expanded substitution symbol listing	CG:9-67
.ssnolist	Inhibit expanded substitution symbol listing (default)	CG:9-67
.tab *size*	Set tab size	CG:9-75
.title "*string***"**	Print a title in the listing page heading	CG:9-79
.width *page width*	Set the page width of the source listing	CG:9-49

Table 9–1. Assembler Directives Summary (Continued)

Directives That Reference Other Files		
Mnemonic and Syntax	**Description**	**Page**
.copy [**"**]*filename*[**"**]	Include source statements from another file	CG: 9-29
.def $symbol_1$ [**,** ... **,** $symbol_n$]	Identify one or more symbols that are defined in the current module and may be used in other modules	CG: 9-41
.global $symbol_1$ [**,** ... **,** $symbol_n$]	Identify one or more global (external) symbols	CG: 9-41
.include [**"**]*filename*[**"**]	Include source statements from another file	CG: 9-29
.mlib [**"**]*filename*[**"**]	Define macro library	CG: 9-56
.ref $symbol_1$ [**,** ... **,** $symbol_n$]	Identify one or more symbols that are used in the current module but may be defined in another module	CG: 9-41
.system $symbol_1$ [**,** ... **,** $symbol_n$]	Identify one or more system (visible to both the PP and MP) symbols	CG: 9-73

Conditional Assembly Directives		
Mnemonic and Syntax	**Description**	**Page**
.break [*well-defined expression*]	Optional end of repeatable block assembly	CG: 9-54
.else *well-defined expression*	Optional conditional assembly	CG: 9-46
.elseif *well-defined expression*	Optional conditional assembly	CG: 9-46
.endif	End conditional assembly	CG: 9-46
.endloop	End repeatable block assembly	CG: 9-54
.if *well-defined expression*	Begin conditional assembly	CG: 9-46
.loop [*well-defined expression*]	Begin repeatable block assembly	CG: 9-54

Table 9–1.Assembler Directives Summary (Concluded)

Symbol Directives		
Mnemonic and Syntax	**Description**	**Page**
.access	Define a point of reference in a structure definition (PP only)	CG:9-69
.asg [**"**]*character string*[**"**], *substitution symbol*	Assign a character string to a substitution symbol	CG:9-24
.endstruct	End structure definition	CG:9-69
.endunion	End union definition (PP only)	CG:9-80
.equ/.set	Equate a value with a symbol	CG:9-65
.eval *well-defined expression*, *substitution symbol*	Perform arithmetic on numeric substitution symbols	CG:9-24
.label *symbol*	Define a load-time relocatable label in a section	CG:9-48
.struct	Begin structure definition	CG:9-69
.tag	Assign structure attributes to a label	CG:9-69
.union	Begin union definition (PP only)	CG:9-80

Miscellaneous Directives		
Mnemonic and Syntax	**Description**	**Page**
.end	End program	CG:9-36
.emsg *string*	Send user-defined error messages to the output device	CG:9-34
.mmsg *string*	Send user-defined messages to the output device	CG:9-34
.newblock	Undefine local labels	CG:9-60
.wmsg *string*	Send user-defined warning messages to the output device	CG:9-34

9.2 Directives That Define Sections

These directives associate the various portions of an assembly language program with the appropriate sections:

- **.bss** reserves space in the .bss section for uninitialized variables.
- **.usect** reserves space in an uninitialized named section. The .usect directive is similar to the .bss directive, but it allows you to reserve space separately from the .bss section.
- **.text** identifies portions of code in the .text section for the MP. The .text section usually contains executable code.
- **.ptext** identifies portions of code in the .ptext section for the PP. The .ptext section usually contains executable code.
- **.data** identifies portions of code in the .data section. The .data section usually contains initialized data.
- **.sect** defines initialized named sections and associates subsequent code or data with that section. A section defined with .sect can contain code or data.

Chapter 12, *Introduction to Common Object File Format,* discusses COFF sections in detail.

Example 9–1 is an output listing that shows how you can use sections directives to associate code and data with the proper sections. Column 1 shows line numbers, and column 2 shows section program counter (SPC) values. (Each section has its own SPC.) When you first place code in a section, its SPC equals 0. When you resume assembling into a section after switching to another section, the SPC of the first section resumes counting as if there had been no intervening code.

The directives in Example 9–1 perform the following tasks:

.text	Initializes five words containing code and directives.
.data	Initializes six words with data values 9h, 10h, 13h, 14h, 15h, and 16h.
.sect "*vardefs***"**	Initializes two words with data values 17h and 18h.
.bss	Reserves 49 uninitialized bytes.
.usect	Reserves 20 uninitialized bytes.

Note that the .bss and .usect directives do not end the current section or begin new sections; they reserve the specified amount of space, and then the assembler resumes assembling code or data into the current section.

Example 9–1. Sections Directives (Example Using MP)

```
 1                      ********************************************
 2                      **  Reserve space in .bss for a variable    **
 3                      ********************************************
 4 00000000                     .bss            numbufs, 30
 5
 6                      ********************************************
 7                      **  Start assembling into the .text section  **
 8                      ********************************************
 9 00000000                     .text
10 00000000   4A11000A mpy:    LD              0ah(r8),r9
11 00000004   426C0123 mloop:  ADD             0x123,r9,r8
12 00000008   E2247FFF         BBZ             mloop,r8,3
13                      ********************************************
14                      **  Start assembling into the .data section  **
15                      ********************************************
16 00000000                     .data
17 00000000   00000009         .word           9, 10
   00000004   0000000A
18                      ********************************************
19                      **      Start assembling into a named,      **
20                      **      initialized section, var_defs       **
21                      ********************************************
22 00000000                     .sect           ”var_defs”
23 00000000   00000011         .word           17, 18
   00000004   00000012
24
25                      ********************************************
26                      **  Resume assembling into the .data section **
27                      ********************************************
28 00000008                     .data
29 00000008   0000000D         .word           13, 14
   0000000C   0000000E
30 0000001E                     .bss            sym, 19 ; Reserve space
31                                                      ; in .bss
32 00000010   0000000F         .word           15, 16  ; Still in .data
   00000014   00000010
33
34                      ********************************************
35                      **       Assemble more code into .text      **
36                      ********************************************
37 0000000C                     .text
38 0000000C   00000005         .word           5, 6
   00000010   00000006
39 00000000            usym    .usect          ”xy”, 20 ; Reserve space
40                                                      ; in xy
41 00000014   00000007         .word           7, 8     ; still in .text
   00000018   00000008
42                              .end
```

Col 1 Col 2 Col 3 Col 4

9.3 Directives That Define Data

The following directives initialize constants for the current section:

- **.byte**, **.ubyte**, **.char**, and **.uchar** place one or more 8-bit values into consecutive bytes of the current section. (All these directives produce identical results except when used in structures. For more information, see Section 9.8, *Symbol Directives*.)

- **.half**, **.uhalf**, **.short**, and **.ushort** place one or more 16-bit values into consecutive 2-byte slots of the current section. The first half-word will be aligned on a 2-byte boundary. (All these directives produce identical results except when used in structures. For more information, see Section 9.8, *Symbol Directives*.)

- **.int, .uint, .word**, **.uword**, **.long**, and **.ulong** place one or more 32-bit values into consecutive 4-byte slots in the current section. The first word will be aligned on a 4-byte boundary. (All these directives produce identical results except when used in structures. For more information, see Section 9.8, *Symbol Directives*.)

Note: The Above Data Defining Directives Accept String Constant Operands.

The .byte, .char, .ubyte, and .uchar directives function exactly the same as .string does. The .half, .uhalf, .short, and .ushort directives place one character in each 2-byte slot. The .int, .uint, .word, .uword, .long, and .ulong directives place one character in each 4-byte slot.

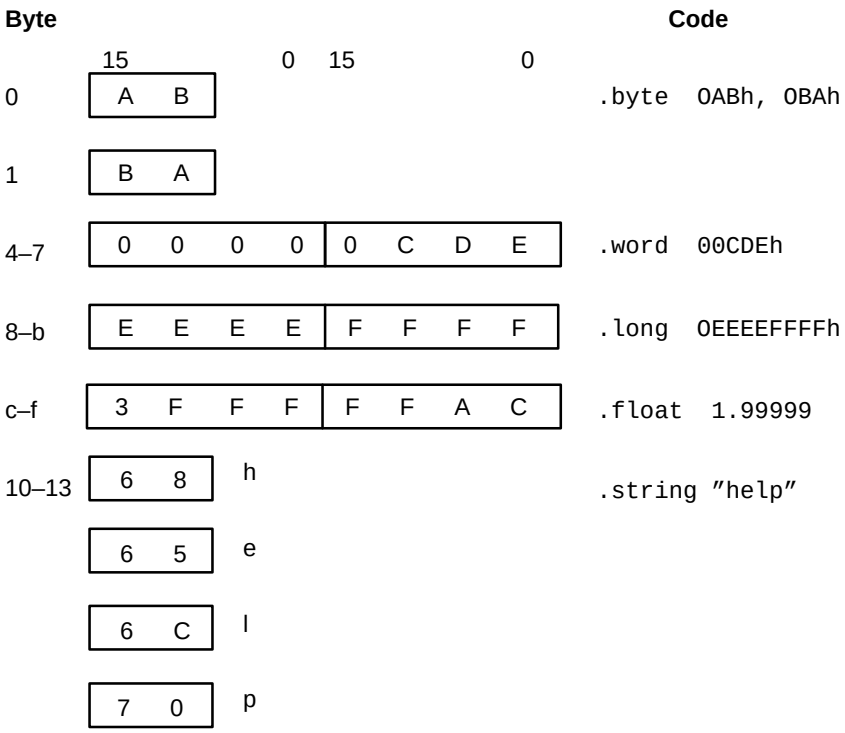
- **.string** places 8-bit characters from one or more character strings into the current section. This directive has the same functionality as the .byte directive. One character is packed into each byte of memory.

- **.float** and **.double** for the PP calculate the single-precision (32-bit) IEEE floating-point representation of a single floating-point value and store it in a 4-byte slot in the current section. For the MP, .float is identical to the PP version, but .double calculates a double-precision (64-bit) value.

Figure 9–1 compares the .byte, .long, .float, .word, and .string directives. For this example, assume the following code has been assembled:

```
1
2                ** Example of initialization directives (MP)
3
4  00000000   AB                .byte           0ABh, 0BAh
   00000001   BA
5  00000004   00000CDE          .word           0CDEh
6  00000008   EEEEFFFF          .long           0EEEEFFFFh
7  0000000C   3FFFFFAC          .float          1.99999
8  00000010   68                .string         ”help”
   00000011   65
   00000012   6C
   00000013   70
```

Figure 9–1. Initialization Directives

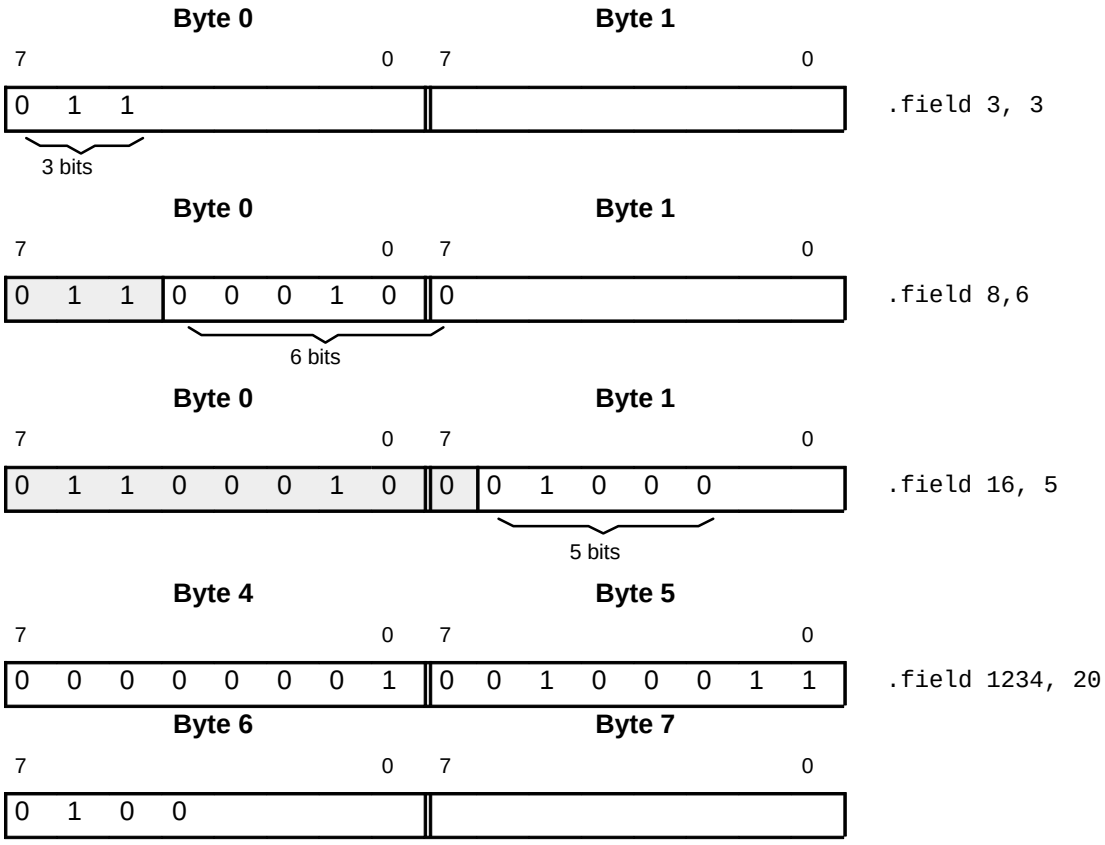


- The **.field** directive places a single value into a specified number of bits in the current word. With a .field directive, you can pack multiple fields into a single word; the assembler does not increment the SPC until a word is filled.

  Figure 9–2 shows a big endian example of how fields are packed into a word. For this example, assume the following code has been assembled. Notice that the SPC doesn't always change (the fields are packed into the same word if the bit counts allow). Note that when the last .field directive is encountered, there are only a total of 18 bits left in bytes 0–3, not enough for a 20-bit field, so the SPC increments to 4:

```
1
2                                ** Example of .field directive (PP)
3
4 00000000 0000000060000000          .field          3, 3
5 00000000 0000000064000000          .field          8, 6
6 00000000 0000000064400000          .field          16, 5
7 00000004 0000000001234000          .field          1234h,20
```

Figure 9–2. The .field Directive



- The **.bes** and **.space** directives reserve a specified number of bytes in the current section. The assembler fills these reserved bytes with 0s.

  Labels are assigned in this manner:

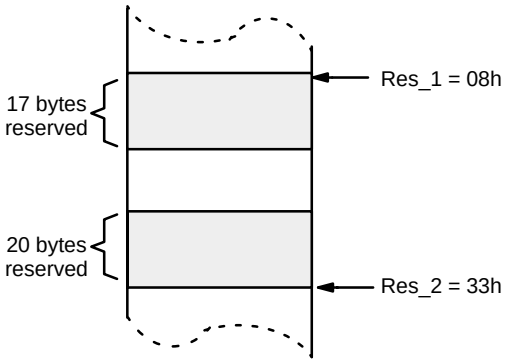
  - When you use a label with .space, the label is assigned the address of the *first* byte that contains reserved space.
  - When you use a label with .bes, the label is assigned the address of the *last* byte that contains reserved space.

  Figure 9–3 shows the .space and .bes directives. Assume the following code has been assembled for this example:

```
1
2                                ** Example of .space and .bes (PP)
3
4 00000000 0000000000000100              .word           100h, 200h
  00000004 0000000000000200
5 00000008                       Res_1:  .space          17
6 0000001c 000000000000000f              .word           15
7 00000033                       Res_2:  .bes            20
8 00000034 00000000000000ba              .byte           0BAh
```

  Res_1 points to the first word in the space reserved by .space. Res_2 points to the last word in the space reserved by .bes.

Figure 9–3. The .space and .bes Directives



9.4 Directives That Align the Section Program Counter

The **.align** directive aligns the SPC at an x-byte boundary, with x being a power of 2. If no operand is used with the align directive, the default value is 4. To ensure that the code following the .align directive is actually aligned to the specified boundary, the assembler does two things:

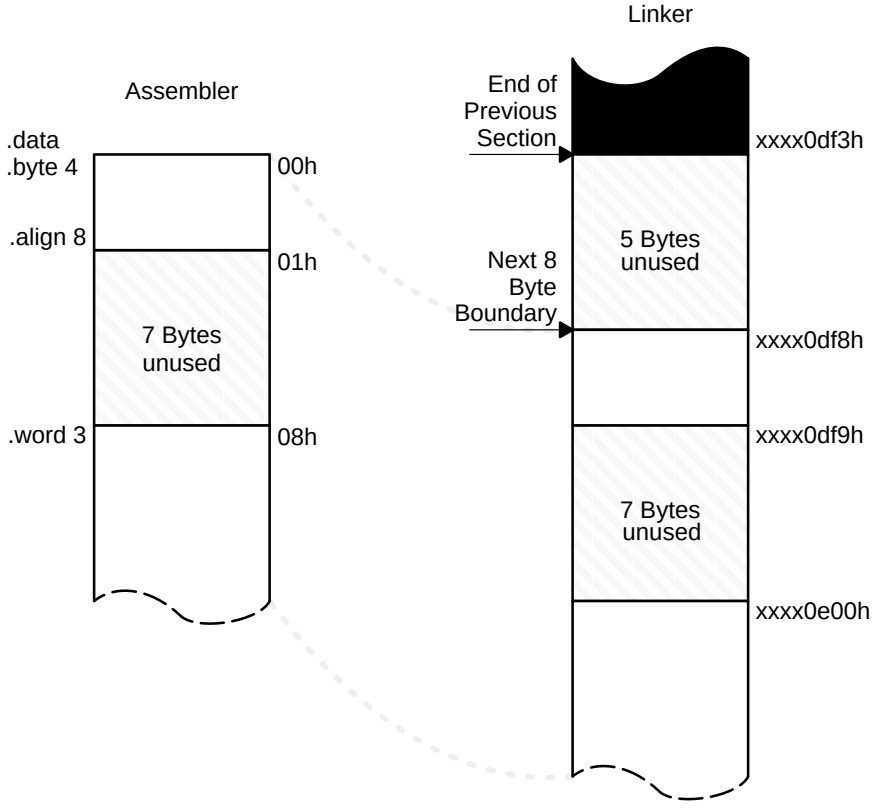
- Increments the SPC to the next x-byte boundary. If the SPC is already aligned at the desired boundary, there is no increment. If no x is specified, the SPC is aligned to a 4-byte boundary.
- Sets a flag that forces the linker to align the section so that the individual alignments within the section are preserved.

For example, assume that the following code is the first code that the linker will place in the .data section:

```
1 00000000                              .data
2 00000000 0000000000000004             .byte 4
3                                       .align 8
4 00000008 0000000000000003             .word 3
```

If the 8-byte alignment is the largest alignment used in the .data section, the linker would align the entire .data section to begin on an 8-byte boundary, as shown in Figure 9–4.

Figure 9–4. The .align Directive



9.5 Directives That Format the Output Listing

The following directives format the listing file:

- The **.drlist** directive causes printing of certain directive lines to the listing, and **.drnolist** turns this printing off.

  The following directives are affected:

.asg	.fclist	.mlist	.var
.break	.fcnolist	.mnolist	.width
.emsg	.length	.sslist	.wmsg
.eval	.mmsg	.ssnolist	

  By default, the assembler acts as if a .drlist had been specified.

- The **.fclist** and **.fcnolist** directives turn on and off the listing of false conditional blocks, which do not generate object code. You can use the .fclist directive to list false conditional blocks exactly as they appear in the source code. You can use the .fcnolist directive to list only the conditional blocks that are actually assembled. If neither directive is specified, the system behaves as if .fclist had been specified.

- The **.length** directive controls the page length of the listing file. You can use this directive to adjust listings for various output devices. Default length is 60 lines.

- The **.width** directive controls the page width of the listing file. You can use this directive to adjust listings for various output devices. Default width is 80 characters.

- The **.list** and **.nolist** directives turn the output listing on and off. You can use the .nolist directive to stop the assembler from printing selected source statements in the listing file. Use the .list directive to turn the listing on again. These directives are not effective unless the assembler command line contains the –l (lowercase L) option.

- The **.mlist** and **.mnolist** directives allow and inhibit macro expansion and loop block listings. You can use the .mlist directive to print all macro expansions and loop blocks to the listing. The .mnolist directive inhibits the listing. The default is .mlist.

- The **.option** directive controls certain features in the listing file. This directive has the following operands:

  A lists all directives and data, performs substitutions, and expands macros.

  B limits the listing of .byte directives to one line.

  D causes printing of certain directives to be turned on:

.asg.	.fclist	.mlist	.var
.break	.fcnolist	.mnolist	.width
.emsg	.length	.sslist	.wmsg
.eval	.mmsg	.ssnolist	

  The .drnolist directive turns the listing off if desired.

  H limits the listing of .half directives to one line.

  L limits the listing of .long directives to one line.

  M turns off macro expansions in the listing.

  N turns off listing (same effect as .nolist directive).

  O turns on listing (same effect as .list directive).

  R resets the B, D, L, M, H, W, and T options.

  T limits the listing of .string directives to one line.

  W limits the listing of .word directives to one line.

  X produces a cross-reference listing of symbols. (You can also obtain a cross-reference listing by invoking the assembler with the –x option.)

- The **.page** directive causes a page eject in the output listing.

- The **.sslist** and **.ssnolist** directives allow and inhibit substitution symbol expansion listing. These directives are useful for debugging the expansion of substitution symbols.

- The **.tab** directive defines tab size.

- The **.title** directive supplies a title that the assembler prints at the top of each page of the listing file.

9.6 Directives That Reference Other Files

These directives supply information for or about other files:

- The **.copy** and **.include** directives tell the assembler to begin reading source statements from another file. When the assembler finishes reading the source statements in the copy/include file, it resumes reading source statements from the current file. The statements read from a copied file (.copy directive) are printed in the listing file (unless .nolist has been specified); the statements read from an included file (.include directive) are *not* printed in the listing file.

- The **.global** directive declares a symbol external so that it is available to other modules at link time. The .global directive does double duty, acting as a .def for defined symbols and as a .ref for undefined symbols. Note that the linker will resolve an undefined global symbol only if it is used in the program. For more information about external symbols, see Section 12.7, *Symbols in a COFF File*.

- The **.system** directive is identical to the .global directive except that a symbol declared with the .system directive can be referenced by both the MP and PP modules.

- The **.def** directive identifies a symbol that is defined in the current module and is available to be used by other modules. The assembler includes the symbol in the symbol table.

- The **.ref** directive identifies a symbol that is used in the current module but that can be defined in another module. The assembler includes the symbol in the symbol table, even if it is not referenced in the current module, so that the linker must resolve its definition.

- The **.mlib** directive supplies the assembler with the name of an archive library that contains macro definitions. When the assembler encounters a macro that is not defined in the current module, it searches for it in the macro library specified with .mlib.

9.7 Conditional Assembly Directives

Conditional assembly directives enable you to assemble certain sections of code according to the evaluation of an expression. Two sets of directives allow you to assemble conditional blocks of code:

- The **.if/.elseif/.else/.endif** directives tell the assembler to assemble a block of code conditionally according to the evaluation of an expression.

.if *expression*	Marks the beginning of a conditional block and assembles code if the .if expression is true (non-zero).
.elseif *expression*	Marks a block of code to be assembled if the .if expression is false and .elseif is true (non-zero).
.else	Marks a block of code to be assembled if the .if expression is false.
.endif	Marks the end of a conditional block and terminates the block.

- The **.loop/.break/.endloop** directives tell the assembler to repeatedly assemble a block of code according to the evaluation of an expression.

.loop [*expression*]	Marks the beginning a repeatable block of code. Continues looping.
.break [*expression*]	Transfers to the code immediately following the .endloop directive when the *expression* is true, or if there is no *expression*. Ignored when the *expression* is false.
.endloop	Marks the end of a repeatable block. Evaluates the loop count, and if the loop count equals the optional *expression* or 1024, executes the code immediately following the .endloop. If the loop count does not equal either quantity, transfers to the code immediately following the matching .loop directive and continues executing.

The assembler supports several relational operators that are useful for conditional expressions. For more information about relational operators, see subsection 8.5.4, *Conditional Expressions*.

9.8 Symbol Directives

These directives equate meaningful symbol names to constant values, strings, structures, and unions.

- The **.asg** directive assigns a character string to a substitution symbol. The value is stored in the substitution symbol table. When the assembler encounters a substitution symbol, it replaces the symbol with its character string value. Substitution symbols can be redefined. For example:

```
.asg  1, x
.byte x
.asg  2, x
.byte x
```

- The **.eval** directive evaluates an expression, translates the results into a ASCII string, and assigns the character string to a substitution symbol. This directive is most useful for manipulating counters within conditionally assembled loops. For example:

```
      .asg     1 , x
      .loop
      .byte    x*10h
      .break   x == 4
      .eval    x+1, x
      .endloop
```

- The **.label** directive defines a special symbol that refers to the loadtime address within the current section. This is useful when a section loads at one address but runs at another address. The .label directive gives you a way to reference the load address of section in the code that moves the section.

- The **.set** and **.equ** directives set a constant value to a symbol. The symbol is stored in the symbol table and cannot be redefined. For example:

```
bval   .set   0100h
       .half  bval, bval*2, bval+12
       add    bval, r0, r1
```

  Note that the .set/.equ directives produce no object code.

- The **.struct/.endstruct** directives set up C-like structure definitions so that similar elements can be grouped together. Element offset calculation is then left up to the assembler.

  The .struct directive begins the structure, and the .endstruct ends it. The .struct/.endstruct directives do not allocate memory. They simply create a symbolic template that can be used repeatedly.

  - The **.tag** directive assigns a label to the structure, creating an instance of the structure. This simplifies the symbolic representation and also provides the ability to define structures that contain other structures. The .tag directive does not allocate memory, and the structure tag (stag) must be defined before it's used. For example:

```
type   .struct            ; structure tag definition
x      .int
y      .int
t_len  .endstruct
coord  .tag  type         ; declare coordinate

       add    coord.y

        .bss  coord, t_len ; actual memory allocation
```

  - The **.access** directive (PP only) defines a point of reference within a structure definition. A structure member's offset is computed relative to the structure's access point. This directive is only used in outermost structure definitions. If it occurs within a nested structure definition, it will be ignored.

Note: The Constant Initializer Directives in a .struct/.endstruct or .union/.endunion Sequence

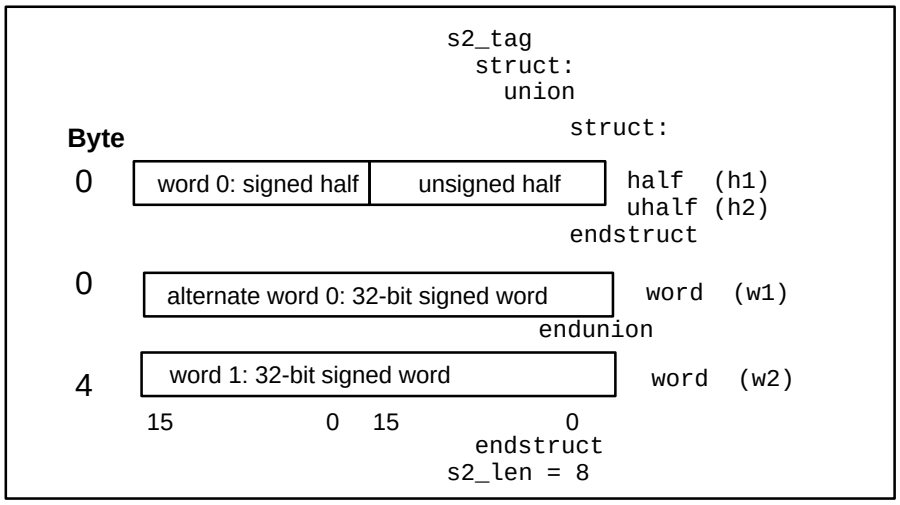
The directives explained in this section *do not* initialize memory when they are part of a .struct/.endstruct or .union/.endunion sequence; rather, they define a member's size. For the PP only, if you load a structure member defined as an unsigned quantity (.ubyte, .uchar, .uhalf, .ushort, .uword, .ulong, or .uint) into a register, the register will be zero extended. If you do the same thing with a signed quantity, the sign will be extended.

- The **.union/.endunion** directives (PP only) create a symbolic template that can be used repeatedly, providing a way to manipulate several different kinds of data in the same storage area. The union sets up a C-like union definition. While it does not allocate any memory, it allows alternate definitions of size and type that may be temporarily stored in the same memory space.

- The **.tag** directive assigns a label to the union, and is associated with the beginning of the union. A union can be defined and given a tag, and later it can be declared as a member of a structure by using the .tag directive. A union may also be declared without a tag, in which case all of its members will be entered in the symbol table, and each member must have a unique name. A union may also be defined within a structure, in which case any reference to such a union will be in association with the structure that encloses it. For example:

```
s2_tag .struct          ; structure tag definition
       .union           ; union is first structure member
       .struct          ; structure is union member
h1     .half            ; h1, h2 and
h2     .uhalf           ; w1 exist in the same
       .endstruct       ; memory
w1     .word            ; word is other union member
       .endunion
w2     .word            ; second structure member
s2_len .endstruct

       .bss   XYZ, s2_len; declare instance of structure

       d7 = *(xba + XYZ.h2); addresses second halfword in
                           ; structure; note type = uh
                           ; so fill with 0s
```

Figure 9–5. Layout of Structure Containing Union Containing Structure



- The **.var** directive defines up to 32 local macro substitution symbols (including parameters) per macro. The .var directive creates temporary substitution symbols with the initial value of the null string. These symbols are not passed in as parameters, and, after expansion, they are lost.

9.9 Miscellaneous Directives

This section discusses miscellaneous directives.

- The **.end** directive terminates assembly. If specified, it is the last source statement of the program processed by the assembler. This directive has the same effect as an end-of-file.
- The **.newblock** directive resets local labels. Local labels are symbols of the form *$n*, where *n* is a decimal digit. They are defined when they appear in the label field. Local labels are temporary labels to be used as operands for jump instructions. The .newblock directive limits the scope of local labels by undefining them after they are used. For more information about local labels, see *Local labels,* page CG:8-13.

These three directives enable you to define your own error and warning messages:

- The **.emsg** directive sends error messages to the standard output device. The .emsg directive generates errors in the same manner that the assembler does, incrementing the error count and preventing the assembler from producing an object file.
- The **.wmsg** directive sends warning messages to the standard output device. The .wmsg directive functions in the same manner that the .emsg directive does, except that it increments the warning count rather than the error count, and it does not prevent the assembler from producing an object file.
- The **.mmsg** directive sends assembly-time messages to the standard output device. The .mmsg directive functions in the same manner that the .emsg and .wmsg directives do, except that it does not affect the error count, the warning count, or prevent the production of the object file.

9.10 Directives Reference

The remainder of this chapter is a reference. Generally, the directives are organized alphabetically, one directive per page; however, related directives (such as .if/.else/.endif) are presented together on one page. Here's an alphabetical table of contents for the directives reference:

Syntax

.align [*size* = 2^n]

Description

The **.align** directive aligns the section program counter (SPC) on the next boundary, depending on the *size* parameter. The *size* may be any power of 2. If no argument is provided to .align, the assembler will align to the next 4-byte boundary.

Operand of 1 aligns SPC to a byte boundary

2 aligns SPC to a half word boundary

4 aligns SPC to a 32-bit word boundary

8 aligns SPC to an instruction boundary (64 bit)

Using the .align directive has two effects:

- The assembler aligns the SPC on an x-byte boundary *within* the current section.
- The assembler sets a flag that forces the linker to align the section so that individual alignments remain intact when a section is loaded into memory.

Example

This example shows several types of alignment including .align 2, .align 4, and a default .align.

```
 1
 2                        ** Examples of .align directive (MP)
 3
 4 00000000                       .data
 5 00000000   04                  .byte          4
 6                                .align         2
 7 00000002   45                  .string        "Errorcnt"
   00000003   72
   00000004   72
   00000005   6F
   00000006   72
   00000007   63
   00000008   6E
   00000009   74
 8 0000000A                       .align
 9 0000000C   60000000            .field         3,3
10 0000000C   6A000000            .field         5,4
11                               .align         2
12 0000000E   6A006000            .field         3,3
13 0000000E                      .align         2
14 00000010   50000000            .field         5,4
15 00000010                      .align
16 00000014   04                  .byte          4
17
```

Syntax

.asg [**"**]*character string*[**"**]**,** *substitution symbol*
.eval *well-defined expression***,** *substitution symbol*

Description

The **.asg** directive assigns character strings to substitution symbols, which are then stored in the substitution symbol table.

The .asg directive can be used in many of the same ways as the .set directive, but while .set assigns a constant value (that cannot be redefined) to a symbol, .asg assigns a character string (that can be redefined) to a substitution symbol.

The **.eval** directive performs arithmetic on substitution symbols; the substitution symbols are stored in the symbol table. This directive evaluates the expression and assigns the ASCII value of the result to the substitution symbol. The .eval directive is especially useful as a counter in .loop/.endloop blocks.

[**"**] *character string*[**"**]	is assigned to the substitution symbol by the assembler. The quotation marks are optional. If there are no quotation marks, the assembler reads characters up to the first comma and removes leading and trailing blanks. In either case, a character string is read and assigned to the substitution symbol.
substitution symbol	is a required parameter that must be a valid symbol name. The substitution symbol may be 32 characters long and must begin with a letter. Remaining characters of the symbol can be a combination of alphanumeric characters, underscores, and dollar signs.
well-defined expression	is an alphanumeric expression consisting of legal values that have been previously defined, so that the result is an absolute expression.

Example This example shows how .asg and .eval can be used.

```
        1                       ; Example of .asg directive (MP)
        2
        3                               .sslist         ; show expanded sub. symbols
        4                               .asg 0,x
        5                               .loop 5
        6                               .eval x+1, x
        7                               .word x
        8                               .endloop
1                                       .eval x+1, x
#                                .eval 0+1, x
1         00000000   00000001           .word x
#                                .word 1
1                                       .eval x+1, x
#                                .eval 1+1, x
1         00000004   00000002           .word x
#                                .word 2
1                                       .eval x+1, x
#                                .eval 2+1, x
1         00000008   00000003           .word x
#                                .word 3
1                                       .eval x+1, x
#                                .eval 3+1, x
1         0000000C   00000004           .word x
#                                .word 4
1                                       .eval x+1, x
#                                .eval 4+1, x
1         00000010   00000005           .word x
#                                .word 5
        9
```

Syntax

.bss *symbol*, *size in bytes* [, *alignment*]

Description

The **.bss** directive reserves space for variables in the .bss section. This directive is commonly used to allocate variables in RAM.

symbol is a required parameter. It defines a symbol that points to the first location reserved by the directive. The symbol name should correspond to the variable that you're reserving space for.

size in bytes is a required parameter; it must be an absolute expression. The assembler allocates *size* bytes in the .bss section. There is no default size.

On the PP only, *size* can also be specified with a structure tag name or a union tag name. In this case the size is equal to the size of the structure or union. If the tag represents a structure with a nonzero access point, then the symbol is associated with the access point value of the structure.

[*alignment*] is an optional parameter that guarantees that the space allocated to the symbol occurs on the specified boundary. This boundary indicates the size of the slot in bytes, and may be set to any power of 2.

If the SPC is already aligned at the specified boundary, it will not be incremented.

Example

This example allocates space for two variables, *temp* and *array*, using the .bss directive. The symbol *temp* points to four words of uninitialized space (at .bss SPC = 0). The symbol *array* points to 100 words of uninitialized space (at .bss SPC = 04h), and this space must be allocated contiguously. Note that symbols declared with the .bss directive can be referenced in the same manner as other symbols and can also be declared external. The label Var_1 marks the beginning of the TEMP block.

```
 1                      ********************************************
 2                      **  Start assembling into .text section (MP) **
 3                      ********************************************
 4 00000000                      .text
 5 00000000   4A3B0007          add r7, r8, r9
 6 00000004   00304009          cmnd r9
 7 00000008   39BA0005          cmp r5, r6, r7
 8                      ********************************************
 9                      **     Allocate 4 bytes in .bss for TEMP     **
10                      ********************************************
11 00000000            Var_1:  .bss    TEMP, 4
12                      ********************************************
13                      **           Still in .text                 **
14                      ********************************************
15 0000000C   327E6008         fdiv.sss r8, r9, r6
16                      ********************************************
17                      ** Allocate 100 bytes in .bss for the symbol **
18                      **               named ARRAY                **
19                      ********************************************
20 00000004                     .bss            ARRAY, 100
21                      ********************************************
22                      **       Assemble more code into .text       **
23                      ********************************************
24 00000010   4A3E4547         fmpy.iii r7, r8, r9
25                      ********************************************
26                      **      Declare external .bss symbols        **
27                      ********************************************
28                              .global         ARRAY, TEMP
29                              .end
```

Syntax

.byte *value*$_1$ [**,** ... **,** *value*$_n$]
.ubyte *value*$_1$ [**,** ... **,** *value*$_n$]
.char *value*$_1$ [**,** ... **,** *value*$_n$]
.uchar *value*$_1$ [**,** ... **,** *value*$_n$]

Description

The **.byte, .ubyte, .char,** and **.uchar** directives place one or more bytes into consecutive words of the current section (they are interchangeable, except when declared in structures.

- An expression that the assembler evaluates and treats as an 8-bit signed or unsigned number, *or*
- A character string enclosed in double quotes. Each character in a string represents a separate value.

Values are not packed or sign-extended; each value or char fills one byte of memory. The assembler truncates values greater than eight bits. You can use as many parameters as you wish, but the total line length cannot exceed 200 characters.

If you use a label, it points to the location where the assembler places the first byte.

Note: .byte, .char, .ubyte, .uchar in a .struct/.endstruct Sequence

Note that when you use these directives in a .struct/.endstruct sequence, they define a member's size; they do not initialize memory. On the PP only, if you define a structure member as signed (.byte or .char) , then when it is loaded into a register sign will be extended. If a structure member is defined as unsigned (.ubyte or .uchar), then when it is loaded into a register it will be zero extended. For more information about .struct/.endstruct, refer to page CG:9-69.

Example

This example places several 8-bit values into consecutive words in memory. The label *strx* has the value 1000h, which is the location of the first initialized word.

```
1
2                              ** Example of .byte directive (PP)
3
4 00000000                             .space          100h * 16
5 00001000 000000000000000a  strx     .byte           10, –1, ”abc”, ’a’
  00001001 00000000000000ff
  00001002 0000000000000061
  00001003 0000000000000062
  00001004 0000000000000063
  00001005 0000000000000061
```

Syntax

.copy [”]*filename*[”]
.include [”]*filename*[”]

(The quote marks surrounding the filename are optional.)

Description

The **.copy** and **.include** directives tell the assembler to read source statements from a different file. The assembler:

1) Stops assembling statements in the current source file.
2) Assembles the statements in the copied/included file.
3) Resumes assembling statements in the main source file, starting with the statement that follows the .copy or .include directive.

The *filename* is a required parameter that names a source file; the filename may be enclosed in double quotes. The filename must follow operating system conventions. You can specify a full pathname (for example, /home/mvp/file1.asm). If you do not specify a full pathname, the assembler searches for the file in:

1) The directory that contains the current source file
2) Any directories named with the –i assembler option
3) Any directories specified by the environment variable A_DIR

For more information about the –i option and A_DIR, see Section 7.3, *Naming Alternate Directories for Assembler Input*.

The statements assembled from a copy file are printed in the assembly listing, unless a .nolist directive has been specified. (See page CG:9-50 for more information about .nolist.) The statements assembled from a file included with the .include directive are never printed in the assembly listing.

The .copy and .include directives can be nested within a file being copied or included. The assembler limits nesting to 32 levels; the host operating system may set additional restrictions. The assembler precedes the line numbers of copied files with a letter code to identify the level of copying. An A indicates the first copied file, B indicates a second copied file, etc.

Example 1 This example uses the .copy directive, which causes the assembler to switch from the current file to read and assemble all source statements from the named file(s) before resuming assembly of the current file.

copy.asm **(source file)**	**byte.asm** **(first copy file)**	**word.asm** **(second copy file)**

`.space 29`
`.copy "byte.asm"`
 	
`**Back in original file`
`.string "done"`
	`** In byte.asm`
	`.byte 32,1+ 'A'`
	`.copy "word.asm"`
	`** Back in byte.asm`
	`.byte 67h + 3q`
		`** In word.asm`
		`.word 0ABCDh, 56q`

Listing file (MP):

```
       1 00000000                       .space          29
       2                                .copy           "byte.asm"
A      1                          **    in byte.asm
A      2 0000001D   20                   .byte   32, 1 + 'A'
         0000001E   42
A      3                                .copy "word.asm"
B      1                          ** in word.asm
B      2
B      3 00000020   0000ABCD             .word   0ABCDh, 56Q
         00000024   0000002E
A      4                          **    back in byte.asm
A      5 00000028   6A                   .byte 67h + 3
A      6
       3                          **    back in original file
       4 00000029   44                   .string "Done"
         0000002A   6F
         0000002B   6E
         0000002C   65
```

Example 2 This example uses the .include directive to cause the assembler to switch from the current file to read and assemble all source statements from the named file(s) before resuming assembly of the current file.

include.asm **(source file)**	**byte2.asm** **(first include file)**	**word2.asm** **(second include file)**

`.space 29`
`.include "byte2.asm"`
 	
`**Back in original file`
`.string "done"`
	`** In byte2.asm`
	`.byte 32,1+ 'A'`
	`.include "word2.asm"`
	`** Back in byte2.asm`
	`.byte 67h + 3q`
		`** In word2.asm`
		`.word 0ABCDh, 56q`

Listing file (MP):

```
       1 00000000                       .space          29
       2                                .include        "byte2.asm"
       3                          **    back in original file
       4 00000029   44                   .string "Done"
         0000002A   6F
         0000002B   6E
         0000002C   65
```

Syntax **.data**

Description The **.data** directive tells the assembler to begin assembling source code into the .data section; .data becomes the current section. The .data section is normally used to contain tables of data or preinitialized variables.

Note that the assembler assumes that .text for the MP and .ptext for the PP are the default sections. Therefore, at the beginning of an assembly, the assembler assembles code into the .text or .ptext section unless you use a section control directive.

Example This PP example illustrates assembly into the .data and .ptext sections.

```
 1
 2                          ********************************************
 3                          **  Start assembling into the .ptext section **
 4                          ********************************************
 5 00000000                         .ptext
 6          0000000000000000 Index   .set    0
 7 00000000 9787000000104100        d7 = Index
                            ********************************************
 8                          **  Start assembling into the .data section  **
 9                          ********************************************
10 00000000                         .data
11 00000000 0000000000000009        .word           9, 10
   00000004 000000000000000a
12 00000008 000000000000000b        .word           11, 12
   0000000c 000000000000000c
13
14                          ********************************************
15                          **      Start assembling into a named,       **
16                          **      initialized section, var_defs        **
17                          ********************************************
18 00000000                         .sect           ”var_defs”
19 00000000 0000000000000011        .word           17, 18
   00000004 0000000000000012
20
21                          ********************************************
22                          **  Resume assembling into the .data section **
23                          ********************************************
24 00000010                         .data
25 00000010 000000000000000d        .word           13, 14
   00000014 000000000000000e
26 00000000                         .bss            sym, 19 ; Reserve space
27                                                          ; in .bss
28 00000018 000000000000000f        .word           15, 16  ; Still .data
   0000001c 0000000000000010
29
30                          ********************************************
31                          **       Assemble more code into .ptext      **
32                          ********************************************
33 00000008                         .ptext
34 00000008 fa87910000104100        d7      =       d2 + d4
35 00000000                 usym    .usect          ”xy”, 20 ; Reserve space
36                                                          ; in xy
37 00000010 0000000000000007        .word           7, 8     ; still .ptext
   00000014 0000000000000008
38                                  .end
```

Syntax

```
.drlist
.drnolist
```

Description

Two directives provide you with the ability to control the printing of assembler directives to the listing file:

- The **.drlist** directive enables the printing of all directives to the listing file.
- The **.drnolist** directive suppresses the printing of the following directives to the listing file:

.asg	.fcnolist	.sslist
.break	.length	.ssnolist
.emsg	.mlist	.var
.eval	.mmsg	.wmsg
.fclist	.mnolist	.width

By default, the assembler acts as if .drlist had been specified.

Example

This example shows how .drnolist inhibits the listing of the above named directives.

Source file:

```
*
*   .drlist/.drnolist example
*
        .asg    0, x
        .loop   2
        .eval   x+1, x
        .endloop

        .drnolist

        .asg    1, x
        .loop   3
        .eval   x+1, x
        .endloop
```

Listing file:

```
      1                         *
      2                         *
      3                         *
      4                         *
      5                         *       .drlist/.drnolist example
      6                         *
      7
      8                                 .asg            0, x
      9                                 .loop           2
     10                                 .eval           x + 1, x
     11                                 .endloop
1                                       .eval           0 + 1, x
1                                       .eval           1 + 1, x
     12
     13                                 .drnolist
     14
     16                                 .loop           3
     17                                 .eval           x + 1, x
     18                                 .endloop
```

Syntax

.emsg *string*
.mmsg *string*
.wmsg *string*

Description

These directives allow you to define your own error and warning messages. The assembler tracks the number of errors and warnings it encounters and prints these numbers on the last line of the listing file.

- The **.emsg** directive sends error messages to the standard output device. The .emsg directive generates errors in the same manner as the assembler does. It increments the error count and prevents the assembler from producing an object file.

- The **.wmsg** directive sends warning messages to the standard output device. The .wmsg directive functions in the same manner as the .emsg directive does, but it increments the warning count rather than the error count, and does not prevent the assembler from producing an object file.

- The **.mmsg** directive sends assembly-time messages to the standard output device. The .mmsg directive functions in the same manner as the .emsg and .wmsg directives do, except that .mmsg does not set the error count or the warning count, and does not prevent the assembler from producing an object file.

Example

In this MP example, the macro MSG_EX requires one parameter. The first time the macro is invoked it has a parameter, PARAM, and it assembles normally. The second time, the parameter is missing and an error is generated.

Source file:

```
          .global     PARAM
MSG_EX    .macro parm1
          .if    $$symlen(parm1) == 0
          .emsg  "ERROR -- MISSING PARAMETER"
          .else
           add   parm1, r7, r8
          .endif
          .endm

          MSG_EX PARAM

          MSG_EX
```

Listing file:

```
        1                                   .global     PARAM
        2                       MSG_EX      .macro parm1
        3                                   .if     $$symlen(parm1) == 0
        4                                   .emsg   ”ERROR –– MISSING PARAMETER”
        5                                   .else
        6                                    add    parm1, r7, r8
        7                                   .endif
        8                                   .endm
        9
       10 00000000                          MSG_EX PARAM
1                                           .if     $$symlen(parm1) == 0
1                                           .emsg   ”ERROR –– MISSING PARAMETER”
1                                           .else
1          00000000   41FB1000              add    PARAM, r7, r8
           00000004   00000000
1                                           .endif
       11
       12 00000008                          MSG_EX
1                                           .if     $$symlen(parm1) == 0
1                                           .emsg   ”ERROR –– MISSING PARAMETER”
 ********* USER ERROR  –  ERROR –– MISSING PARAMETER
1                                           .else
1                                            add    parm1, r7, r8
1                                           .endif
       13

 1 Error,  No Warnings
```

In addition, the following messages are sent to standard output by the assembler:

```
MVP MP Macro Assembler     Version x.xx
Copyright (c) 1993–1995    Texas Instruments Incorporated
 PASS 1
 PASS 2
”emsg.asm” line 12: ** USER ERROR **
  ERROR –– MISSING PARAMETER

 1 Error,  No Warnings

Errors in source – Assembler Aborted
```

Syntax

```
.end
```

Description

The **.end** directive is an optional directive that terminates assembly. It will be the last source statement of a program processed by the assembler. The assembler ignores any source statements that follow a .end directive.

This directive has the same effect as an end-of-file. You can use .end when you're debugging and you'd like to stop assembling at a specific point in your code.

> **Note: Ending a Macro**
>
> Use .endm to end a macro.

Example

This example shows how the .end directive terminates assembly. The example is similar to the .emsg example on the preceding pages, except that a .end directive precedes the second invocation of the MSG_EX macro. The assembler ignores the second MSG_EX macro.

Source file:

```
          .global     PARAM
MSG_EX    .macro parm1
          .if    $$symlen(parm1) = 0
          .emsg  "ERROR -- MISSING PARAMETER"
          .else
           add   parm1, r7, r8
          .endif
          .endm
          MSG_EX PARAM
          .end
          MSG_EX
```

Listing file:

```
        2                       **  example of .emsg directive
        3
        4                               .global PARAM
        5
        6                       MSG_EX  .macro  ADDRA
        7                               .if     $$symlen(ADDRA) = 0
        8                               .emsg   "ERROR -- MISSING PARAMETER"
        9                               .else
       10                                ADD    R2,R3,R7
       11                               .endif
       12
       13                               .endm
       14
       15 00000000                      MSG_EX  PARAM
1                                       .if     $$symlen(ADDRA) = 0
1                                       .emsg   "ERROR -- MISSING PARAMETER"
1                                       .else
1          00000000   38FB0002          ADD    R2,R3,R7
1                                       .endif
1
       16                               .end

 No Errors,  No Warnings
```

Syntax

```
.fclist
.fcnolist
```

Description

Two directives enable you to control the listing of false conditional blocks:

- The **.fclist** directive allows the listing of conditional blocks that do not produce code (false blocks). By default, the assembler behaves as if you had used .fclist.
- The **.fcnolist** directive inhibits the listing of false conditional blocks that do not produce code. Only code that actually assembles in the conditional block appears in the listing. The .if, .elseif, .else, and .endif directives do not appear.

Example

This example shows the assembly language and listing files for code with and without the conditional blocks listed. This is the unassembled file:

Source file:

```
AAA    .set   1
BBB    .set   0
       .fclist

       .if       AAA
       ADD    1024, R2, R3
       .else
       ADD    1024*10, R2, R3
       .endif

       .fcnolist

       .if       AAA
       ADD    1024, R2, R3
       .else
       ADD    1024*10, R2, R3
       .endif
```

Listing file:

```
 1                        **      example of .fclist/.fcnolist directives
 2
 3              00000001 AAA     .set    1
 4              00000000 BBB     .set    0
 5                               .fclist
 6
 7                               .if     AAA
 8  00000000    18AC0400         ADD     1024,R2,R3
 9                               .else
10                               ADD     1024 * 10, r2, R3
11                               .endif
12
13                               .fcnolist
14
16  00000004    18AC0400         ADD     1024, R2, R3
```

Syntax

.field *value* [*, size in bits*]

Description

The **.field** directive initializes multiple-bit fields within a 4-byte slot of memory. This directive has two operands:

value is a required parameter. It is an expression that is evaluated and placed in the field. If the value is relocatable, the *size in bits* must be 32.

[*size in bits*] is an optional parameter. It specifies a number from 1–32, which is the number of bits in the field. If you do not specify a size, the assembler assumes that the size is 32 bits. If you specify a value that cannot fit in *size* bits, the assembler truncates the value and issues a warning message. For example, .field 3,1 causes the assembler to truncate the value 3 to 1; the assembler also prints the message:

```
WARNING! line xx: W0001: Field value truncated to xx
         .field xxxxh, x
```

Successive .field directives pack values into the specified number of bits in the current 32-bit slot. Fields are packed starting at the most significant bit, moving toward the least significant bit as more fields are added. If the assembler encounters a field size that does not fit in the current 32-bit word, it writes out the word, advances the SPC to the next word boundary, and begins packing fields into the next word.

You can use the .align directive to force the next .field directive to begin packing into a new word.

If you use a label, it points to the LSbyte of the word that contains the specified field.

Note also that when you use .field in a .struct/.endstruct sequence, .field defines a member's size; it does not initialize memory. For more information about .struct/.endstruct, refer to Section 9.8.

The .field directive works differently depending on the endianess of the code.

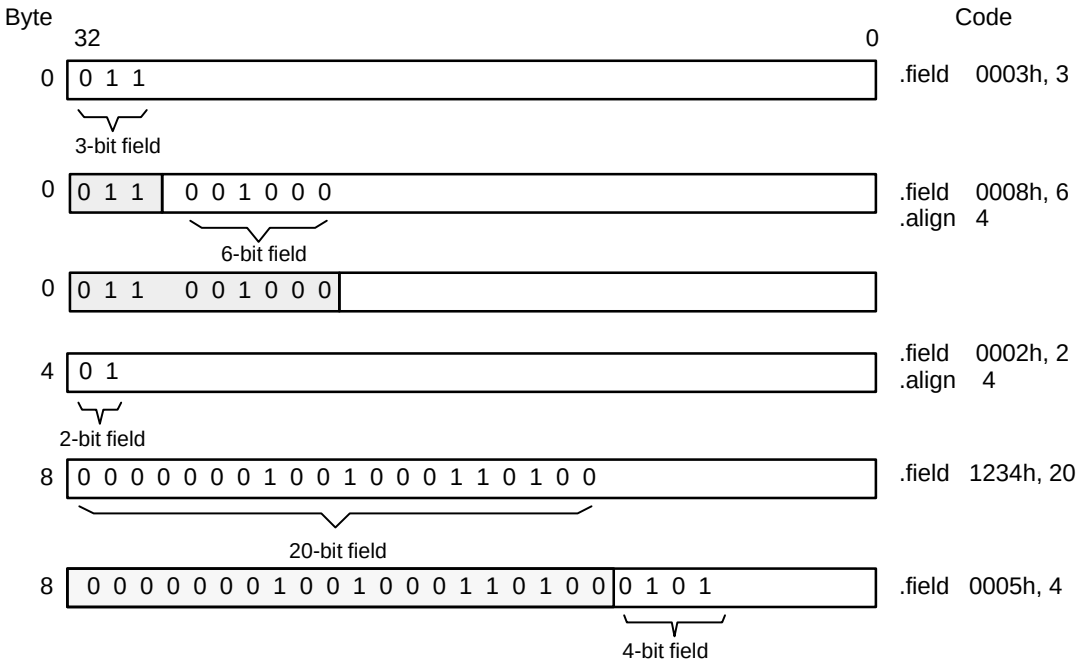
Example

This example shows how fields are packed into a word. Notice that the SPC does not change until a word is filled and the next word is begun.

```
 1                      ****************************************
 2                      *       Example of .field directive (MP)  *
 3                      *       Initialize a 3 – bit field        *
 4                      ****************************************
 5 00000000   60000000          .field  03h, 3
 6                      ****************************************
 7                      *       Initialize a 6 – bit field        *
 8                      ****************************************
 9 00000000   64000000          .field  08h, 6
10                      ****************************************
11                      *           Write out the word           **
12                      ****************************************
13 00000001                    .align 4
14                      ****************************************
15                      *       This field starts at byte 4      **
16                      ****************************************
17 00000004   80000000         .field 02h, 2
18 00000004                    .align 4
19                      ****************************************
20                      *       Initialize a 20-bit field         *
21                      ****************************************
22 00000008   01234000         .field  1234h, 20
23                      ****************************************
24                      *       This field is in next long word   *
25                      ****************************************
26 0000000A   01234500         .field  05h, 4
```

Figure 9–6 shows how the directives in this example affect memory.

Figure 9–6. The .field Directive



Syntax

.float *value* [**,** *...* **,** *value*$_n$]
.double *value* [**,** *...* **,** *value*$_n$]

Description

The **.float** and **.double** directives for the PP place the floating-point representation of a single precision floating-point constant into a word in the current section. The value must be a floating-point constant or a symbol that has been equated to a floating-point constant. Each constant is converted to a floating-point value in IEEE single-precision (32-bit) format. For the MP, the .float directive is identical, but the .double directive produces IEEE double-precision values in double-precision (64-bit) format.

Note that when you use .float in a .struct/.endstruct sequence, .float defines a member's size; it does not initialize memory. For more information about .struct/.endstruct, refer to Section 9.8.

Example

The following are some examples of the .float directive:

```
1                       **      .float directive (PP)

2
3 00000000 00000000e9045951           .float  –1.0e25
4 00000004 0000000040400000           .float  3
5 00000008 0000000042f60000           .float  123
```

Syntax

.global $symbol_1$ [**,** ... **,** $symbol_n$]
.def $symbol_1$ [**,** ... **,** $symbol_n$]
.ref $symbol_1$ [**,** ... **,** $symbol_n$]

Description

These directives identify global symbols, which are defined externally or can be referenced externally:

- The **.def** directive identifies a symbol that must be defined in the current module and may be accessed by other files. The assembler places this symbol in the symbol table of the current module.
- The **.ref** directive identifies a symbol that may be used in the current module, but may be defined in another module. The linker must resolve this symbol's definition at link time.
- The **.global** directive acts as a .ref or a .def, as needed. The linker resolves this symbol's definition at link time and creates a symbol table entry if it is used in the module.

A global *symbol* is defined in the same manner as any other symbol; that is, it appears as a label or is defined by the .set, .bss, or .usect directive. If a global symbol is defined more than once, the linker issues a multiple-definition error, as would happen with any symbol. Note that .ref always creates a symbol table entry for a symbol, whether the module uses the symbol or not; .global, however, creates an entry only if the module actually uses the symbol. To save space in the symbol table, use .global.

A symbol may be declared global for two reasons:

- If the symbol is not defined in the current module (including macro, copy, and include files), the .global or .ref directive tells the assembler that the symbol is defined in an external module. This prevents the assembler from issuing an unresolved reference error. At link time, the linker looks for the symbol's definition in other modules.
- If the symbol is defined in the current module, the .global or .def directive declares that the symbol and its definition can be used externally by other modules. These references are also resolved at link time.

Example This example uses four files:

- file1.lst and file3.lst are similar. Both files define the symbol Init and make it available to other modules; both files use the external symbols x, y, and z. file1.lst uses the .global directive to identify these global symbols; file3.lst uses .ref and .def to identify the symbols. In file1, y and z don't appear in the symbol table of the module because they are not referenced.
- file2.lst and file4.lst are equivalent. Both files define the symbols x, y, and z and make them available to other modules; both files use the external symbol Init. file2.lst uses the .global directive to identify these global symbols; file4.lst uses .ref and .def to identify the symbols.

file1.lst:

```
 1                      ;**  symbols defined/identified wt .global
 2
 3                      ; Global symbol defined in this file
 4                              .global INIT
 5
 6                      ; Global symbols defined in file2.lst
 7                              .global  X, Y, Z
 8
 9 00000000             INIT:
10 00000000   28EC0056          ADD     0x0056, r3,r5
11 00000004   00000000!         .word   X
12                      ;       .
13                      ;       .
14                      ;       .
15                              .end
```

file2.lst:

```
 1                      **   symbols defined/identified wt .global
 2
 3                      ; Global symbol defined in this file
 4                              .global X, Y, Z
 5                      ; Global symbols defined in file1.lst
 6                              .global  INIT
 7             00000001 X:      .set    1
 8             00000002 Y:      .set    2
 9             00000003 Z:      .set    3
10
11 00000000   00000000!         .word   INIT
12                      ;       .
13                      ;       .
14                      ;       .
15                              .end
```

file3.lst:

```
 1                      **  symbols defined/identified wt .def/.ref
 2
 3                      ; Global symbol defined in this file
 4                              .def INIT
 5                      ; Global symbols defined in file4.lst
 6                              .ref    X, Y, Z
 7
 8 00000000             INIT:
 9
10 00000000   28EC0056         ADD  0x00000056,r3,r5
11
12 00000004   00000000!        .word   X
13                      ;       .
14                      ;       .
15                      ;       .
16                              .end
```

file4.lst:

```
 1                      ** symbols defined/identified with .def/.ref
 2
 3                      ; Global symbol defined in this file
 4                              .def    X, Y, Z
 5                      ; Global symbols defined in file3.lst
 6                              .ref    INIT
 7
 8             00000001 X:      .set    1
 9             00000002 Y:      .set    2
10             00000003 Z:      .set    3
11 00000000   00000000!        .word   INIT
12                      ;       .
13                      ;       .
14                      ;       .
15                              .end
```

Syntax

.**half** *value*$_1$ [**,** ... **,** *value*$_n$]
.**uhalf** *value*$_1$ [**,** ... **,** *value*$_n$]
.**short** *value*$_1$ [**,** ... **,** *value*$_n$]
.**ushort** *value*$_1$ [**,** ... **,** *value*$_n$]

Description

The **.half**, **.uhalf**, **.short**, and **.ushort** directives place one or more values into consecutive 16-bit fields in the current section. (These directives are interchangeable, except when used in a structure declaration. See note below.) Each value is placed in a 2-byte slot by itself. A *value* can be either:

- An expression that the assembler evaluates and treats as a 16-bit signed or unsigned number, or
- A character string enclosed in double quotes. Each character in a string represents a separate value and is stored alone in the least significant 8 bits of a 16-bit field, which is padded with 0s.

You can use as many values as fit on a single line (the assembler reads a maximum of 200 characters). If you use a label, it points to the first half-word that is initialized.

These directives perform a 2-byte alignment before any data is written to the section. This guarantees that data will reside on a 2-byte boundary.

Note: .half, .short, .uhalf, .ushort in a .struct/.endstruct Sequence

Note that when you use these directives in a .struct/.endstruct sequence, they define a member's size; they do not initialize memory. For the PP only, if you define a structure member as signed (.half or .short) , then when it is loaded into a register sign will be extended. If a structure member is defined as unsigned (.uhalf or .ushort), then when it is loaded into a register zero extended. For more information about .struct/.endstruct, refer to page CG:9-69.

Example This example uses the .short, .ushort, .half, and .uhalf directives to initialize 2-byte slots:

```
3 00000000 00000000000000ab  DAT1:    .half   0ABh, 'A' + 100h, 'g'
  00000002 0000000000000141
  00000004 0000000000000067
4 00000006 00000000000000ab           .uhalf  0ABh, 'A' + 100h, 'g'
  00000008 0000000000000141
  0000000a 0000000000000067
5 0000000c 000000000000000a           .short  10, -1, 35 + 'a'
  0000000e 0000000000000ffff
  00000010 0000000000000084
6 00000012 000000000000000a           .ushort 10, -1, 35 + 'a'
  00000014 0000000000000ffff
  00000016 0000000000000084
```

Syntax

.if *well-defined expression*
assemble code block when the expression is true

.elseif *well-defined expression*
assemble code block when the .if expression is false and the .elseif expression is true

.else
assemble code block when all .if and .elseif expressions are false

.endif
terminate condition block

Description

These directives provide conditional assembly:

- The **.if** directive marks the beginning of a conditional block. The *well-defined expression* is a required parameter.
  - If the expression evaluates to true (nonzero), the assembler assembles the code that follows it (until it encounters a .elseif, .else, or .endif directive).
  - If the expression evaluates to false (0), the assembler assembles code that follows a .elseif (if present), .else (if present), or .endif (if no .elseif or .else is present).
- The **.elseif** directive identifies a block of code to be assembled when the .if expression is false (0) and the .elseif expression is true (nonzero). When the .elseif expression is false, the assembler continues to the next .elseif (if present), .else (if present), or .endif. The .elseif directive is optional in the conditional block; if an expression is false and there is no .elseif statement, the assembler continues with the code that follows a .else (if present) or a .endif.
- The **.else** directive identifies a block of code that the assembler assembles when the .if expression is false (0). This directive is optional in the conditional block; if an expression is false and there is no .else statement, the assembler continues with the code that follows the .endif.
- The **.endif** directive terminates a conditional .if block.

The .elseif and .else directives can be used in the same conditional assembly block and .elseif can be used more than once within a conditional assembly block.

For information about relational operators recognized by the assembler, refer to subsection 8.5.4, *Conditional Expressions*.

Example The following are some examples of conditional assembly using the PP assembler:

```
 1          0000000100000001  SYM1    .set    1
 2          0000000200000002  SYM2    .set    2
 3          0000000300000003  SYM3    .set    3
 4          0000000400000004  SYM4    .set    4
 5
 6                            If_4:   .if     SYM4 == SYM2 * SYM2
 7 00000000 0000000000000004          .byte   SYM4    ; Equal values
 8                                    .else
 9                                    .byte   SYM2 * SYM2; Unequal values
10                                    .endif
11
12                            If_5:   .if     SYM1 <= 10
13 00000001 000000000000000a          .byte   10      ; Less than / equal
14                                    .else
15                                    .byte   SYM1    ; Greater than
16                                    .endif
17
18                            If_6:   .if     SYM3 * SYM2 != SYM4 + SYM2
19                                    .byte   SYM3 * SYM2 ; Unequal Value
20                                    .else
21 00000002 0000000000000008          .byte   SYM4 + SYM4 ; Equal Values
22                                    .endif
23
24                            If_7:   .if SYM1 == 2
25                                    .byte   SYM1
26                                    .elseif SYM2 + SYM3 == 5
27 00000003 0000000000000005          .byte   SYM2 + SYM3
28
29                                    .endif
```

Syntax

.label *symbol*

Description

The **.label** directive defines a special *symbol* that refers to the loadtime address, rather than the runtime address, within the current section. (Use the label field to refer to the runtime address.) Most sections created by the assembler have relocatable addresses. The assembler assembles each section as if it started at 0, and the linker relocates it to the address at which it loads and runs.

For some applications, it is desirable to have a section load at one address and run at another address.

Such a section is assigned two addresses at link time: a load address and a run address. All labels defined in the section are relocated to refer to the runtime address so that references to the section (such as branches) are correct when the code runs.

The .label directive creates a special label that refers to the loadtime address. This function is useful primarily to tell the code that relocates the section where the section was loaded.

Example

The following example illustrates the .label directive as used with code that is loaded at one address and run at a second address:

```
;-------------------------------------------
; .label Example
;-------------------------------------------
      .sect ".examp"
      .label examp_load   ; load address of section
start:                    ; run address of section
      <code>
finish:                   ; run address of section end
      .label examp_end    ; load address of section end
```

For more information about assigning runtime and loadtime addresses in the linker, refer to Section 13.8, *Specifying a Section's Runtime Address*.

Syntax

.length *page length*
.width *page width*

Description

The **.length** directive sets the page length of the output listing file. It affects the current and following pages. You can reset the page length with another .length directive.

- Default length: 60 lines
- Minimum length: 1 line
- Maximum length: 32,767 lines

The **.width** directive sets the page width of the output listing file. It affects the next line assembled and the lines following; you can reset the page width with another .width directive.

- Default width: 80 characters
- Minimum width: 80 characters
- Maximum width: 200 characters

Note that the width refers to a full line in a listing file; the line counter value, SPC value, and object code are counted as part of the width of a line. Comments and other portions of a source statement that extend beyond the page width are truncated in the listing.

The assembler does not list the .width and .length directives.

Example

The following example changes the page length and width.

```
*****************************************
**        Page length = 65 lines       **
**        Page width  = 85 characters  **
*****************************************
          .length    65
          .width     85

*****************************************
**        Page length = 55 lines       **
**        Page width  = 100 characters **
*****************************************
          .length    55
          .width     100
```

Syntax

```
.list
.nolist
```

Description

Two directives enable you to control the printing of the source listing:

- The **.list** directive allows the printing of the source listing.
- The **.nolist** directive suppresses the source listing output until a .list directive is encountered. The .nolist directive can be used to reduce assembly time and source listing size. It can be used in macro definitions to inhibit the size of the macro expansion.

The assembler does not print the .list or .nolist directives or the source statements that appear after a .nolist directive. However, it continues to increment the line counter. You can nest the .list/.nolist directives; each .nolist needs a matching .list to restore the listing. At the beginning of an assembly, the assembler acts as if it has assembled a .list directive.

> **Note: Creating a Listing File (–l option)**
>
> If you don't request a listing file when you invoke the assembler, the assembler ignores the .list and .nolist directive.

Example

This example uses the .copy directive to insert source statements from another file. The first time this directive is encountered, the assembler lists the copied source lines in the listing file. The second time this directive is encountered, the assembler does not list the copied source lines, because a .nolist directive was assembled. Note that the .nolist, the second .copy, and .list directives do not appear in the listing file. Note also that the line counter is incremented even when source statements are not listed.

Source file:

```
            .copy    "copy2.asm"
* Back in original file
            .nolist
            .copy    "copy2.asm"
            .list
* Back in original file
            .string  "Done"
```

Listing file (PP):

```
      1                                      .copy "copy2.asm"
A     1                              **      in copy2.asm
A     2 00000000 0000000000000020           .byte   32, 1 + 'A'
        00000001 0000000000000042
A     4
      2                              * Back in original file
      6                              * Back in original file
      7 00000006 0000000000000044           .string "Done"
        00000007 000000000000006f
        00000008 000000000000006e
        00000009 0000000000000065
```

Syntax

.**long** *value*$_1$ [, ... , *value*$_n$]
.**ulong** *value*$_1$ [, ... , *value*$_n$]
.**int** *value*$_1$ [, ... , *value*$_n$]
.**uint** *value*$_1$ [, ... , *value*$_n$]
.**word** *value*$_1$ [, ... , *value*$_n$]
.**uword** *value*$_1$ [, ... , *value*$_n$]

Description

The **.long, .ulong, .int, .uint, .word** and **.uword** directives are interchangeable, except when declared as part of a structure (See note below). All these directives place one or more values into consecutive 32-bit fields in the current section. Each value is placed in a 32-bit word by itself. A *value* can be either:

- An expression that the assembler evaluates and treats as a 32-bit signed or unsigned number, or
- A character string enclosed in double quotes. Each character in a string represents a separate value and is stored alone in the least significant 8 bits of a 32-bit field, which is padded with 0s.

The values can be either absolute or relocatable expressions. If an expression is relocatable, the assembler generates a relocation entry that refers to the appropriate symbol; the linker can then correctly patch (relocate) the reference. This allows you to initialize memory with pointers to variables or labels.

You can use as many values as fit on a single line (200 characters). If you use a label, it points to the first word that is initialized.

The .long, .ulong, .int, .uint, .word, and .uword directives perform a 4-byte alignment before any data is written to the section. This guarantees that data will reside on a 4-byte boundary.

Note: Structure Declarations with .long, .word, .ulong, .uword

Note that when you use these directives in a .struct/.endstruct sequence, they define a member's size; they do not initialize memory. For the PP only, when you define a structure member as a signed quantity (.long, .int, or .word) it will be sign extended when loaded into a register. When you define a structure member as an unsigned quantity (.ulong, .uint, .uword), it will be zero extended when loaded into a register.

Example 1 This example uses the .long directive to initialize words.

```
1                              **      Example of .long directive (PP)
2 00000000 0000000000000ABCD   DAT1:   .long   0ABCDh, 'A' + 100h, 'g', 'o'
  00000004 0000000000000141
  00000008 0000000000000067
  0000000c 000000000000006F
3 00000010 0000000000000018'           .long   DAT2, 0AABBCCDDh
  00000014 00000000AABBCCDD
4 00000018                     DAT2:
```

Example 2 This example uses the .word directive to initialize words.

```
1                              ** Example of .word directive (PP)
2 00000000                             .space  64h
3 00000000                             .bss    PAGE, 128
4 00000080                             .bss    SYMPTR, 3
5 00000068 9B871A4000000056    INST:    d7   = 0x0056
6 00000070 000000000000000A            .word   10,SYMPTR, -1,35 + 'a',INST
  00000074 0000000000000080-
  00000078 00000000FFFFFFFF
  0000007c 0000000000000084
  00000080 0000000000000068'
```

Syntax

.loop [*well-defined expression*]
repeatedly assemble code block
.break [*well-defined expression*]
continue assembling when the .break expression is false (zero); go to code following .endloop if true (nonzero)
.endloop
evaluate loop count, transfer to code following .loop unless loop count evaluates equal

Description

Three directives enable you to repeatedly assemble a block of code:

- The **.loop** directive begins a repeatable block of code. Code immediately following .loop is assembled repeatedly until the loop count is exhausted, then, transfer is made to code immediately following .endloop. If there is no expression, the loop count defaults to 1024.
- The **.break** directive is optional, along with its expression. When the expression is false (0), the .break is ignored. When the expression is true (nonzero), or omitted, the assembler breaks the loop and assembles the code immediately following the .endloop directive.
- The **.endloop** directive terminates a repeatable block of code, and increments the loop count, then evaluates the loop count and transfers to the code following .loop, unless the *well-defined expression* equals the loop count.

Example This example illustrates how these directives can be used with the .eval directive.

```
                1                       **      .loop/.break/.endloop directives
                2
                3                               .eval   0, x                    \
                4                       COEF    .loop ; coefficient table      |
                5                               .word   x * 100                 > loop
                6                               .eval   x + 1, x               |  definition
                7                               .break  x == 7                  |
                8                               .endloop                        /
1               00000000   00000000             .word   0 * 100                 \
1                                                .eval   0 + 1, x               |
1                                                .break  1 ==  7                |
1               00000004   00000064             .word   1 * 100                |
1                                                .eval   1 + 1, x               |
1                                                .break  2 ==  7                |
1               00000008   000000C8             .word   2 * 100                |
1                                                .eval   2 + 1, x               |
1                                                .break  3 ==  7                |
1               0000000C   0000012C             .word   3 * 100                |  loop
1                                                .eval   3 + 1, x               |  iterations
1                                                .break  4 ==  7                |
1               00000010   00000190             .word   4 * 100                |
1                                                .eval   4 + 1, x               |
1                                                .break  5 ==  7                |
1               00000014   000001F4             .word   5 * 100                |
1                                                .eval   5 + 1, x               |
1                                                .break  6 ==  7                |
1               00000018   00000258             .word   6 * 100                |
1                                                .eval   6 + 1, x               |
1                                                .break  7 ==  7                /
```

Syntax

.mlib [”]*filename*[”]

(The quote marks surrounding the filename are optional.)

Description

The **.mlib** directive provides the assembler with the name of a macro library. A macro library is a collection of files that contain macro definitions. These files are bound into a single file (called a library or archive) by the archiver. Each member of a macro library may contain one macro definition that corresponds to the name of the file.

Note: Restrictions When Using Macro Libraries

When using macro libraries, the following rules must always be followed:

- Macro library members must be source files (not object files).
- The filename of a macro library member *must* be the same as the macro name, and its extension *must* be .asm for the MP assembler, .s for the PP.

The filename must follow host operating system conventions; it may be enclosed in double quotes. You can specify a full pathname (for example, /home/mvp/macs.lib). If you do not specify a full pathname, the assembler searches for the file in three places, in the order given:

1) The directory that contains the current source file.
2) Any directories named with the –i assembler option .
3) Any directories specified by the environment variable A_DIR.

For more information about the –i option and the environment variable, see Section 7.3, *Naming Alternate Directories for Assembler Input*.

When the assembler encounters a .mlib directive, it opens the library and creates a table of the library’s contents. The assembler enters the names of the individual library members into the opcode table as library entries; this redefines any existing opcodes or macros that have the same name. If one of these macros is called, the assembler extracts the entry from the library and loads it into the macro table. The assembler expands the library entry in the same way it expands other macros, but it does not place the source code into the listing. Only macros that are actually called from the library are extracted, and they are extracted only once.

Example

This example creates a macro library that defines two macros, inc4 and dec4. The file inc4.asm contains the definition of inc4, and dec4.asm contains the definition of dec4.

inc4.asm

```
 * Macro for incrementing

inc4 .MACRO  reg1, reg2,
             reg3, reg4

       reg1 = reg1 + 1
       reg2 = reg2 + 1
       reg3 = reg3 + 1
       reg4 = reg4 + 1

       .ENDM
```

dec4.asm

```
* Macro for decrementing

dec4   .MACRO reg1, reg2
              reg3, reg4

       reg1 = reg1 – 1
       reg2 = reg2 – 1
       reg3 = reg3 – 1
       reg4 = reg4 – 1

       .ENDM
```

Use the archiver to create a macro library:

```
mvpar –a mac inc4.asm dec4.asm
```

Now you can use the .mlib directive to reference the macro library and define the inc4 and dec4 macros:

```
      1                              **     .mlib directive (PP)
      2
      3                                      .mlib   ”mac.lib”
      4 00000000                             inc4    d1, d2, d3, d4 ; macro call
1       00000000 8a21208000104100             d1 = d1 + 1
1       00000008 8a22408000104100             d2 = d2 + 1
1       00000010 8a23608000104100             d3 = d3 + 1
1       00000018 8a24808000104100             d4 = d4 + 1
      5 00000020                             dec4    d5, d6, d7, d1 ; macro call
1       00000020 982d008000104100             d5 = d5 – 1
1       00000028 982e008000104100             d6 = d6 – 1
1       00000030 982f008000104100             d7 = d7 – 1
1       00000038 9829008000104100             d1 = d1 – 1
```

Syntax

```
.mlist
.mnolist
```

Description

Two directives enable you to control the listing of macro and repeatable block expansions in the listing file:

- The **.mlist** directive allows macro and .loop/.endloop block expansions in the listing file.
- The **.mnolist** directive inhibits macro and .loop/.endloop block expansions in the listing file.

By default, all code encountered in macros and .loop/.endloop blocks is listed.

Example

This example defines a macro named str_3. The first time the macro is called, the macro expansion is listed (by default). The second time the macro is called, the macro expansion is not listed, because a .mnolist directive was assembled. The third time the macro is called, the macro expansion is again listed because a .mlist directive was assembled.

```
        1                       ****************************************
        2                       **    Example of .mlist/.mnolist (MP)   **
        3                       ****************************************
        4
        5                       str_3   .macro  p1,p2,p3
        6                               .string ":p1:", ":p2:", ":p3:"
        7                               .endm
        8
        9 00000000                      str_3   "red", "green", "blue"
1         00000000   72                 .string "red", "green", "blue"
          00000001   65
          00000002   64
          00000003   67
          00000004   72
          00000005   65
          00000006   65
          00000007   6E
          00000008   62
          00000009   6C
          0000000A   75
          0000000B   65
       10
       11                               .mnolist
       12 0000000C                      str_3   "red", "green", "blue"
       13
       14                               .mlist
       15 00000018                      str_3   "red", "green", "blue"
1         00000018   72                 .string "red", "green", "blue"
          00000019   65
          0000001A   64
          0000001B   67
          0000001C   72
          0000001D   65
          0000001E   65
          0000001F   6E
          00000020   62
          00000021   6C
          00000022   75
          00000023   65
```

Syntax

.newblock

Description

The **.newblock** directive undefines any local labels currently defined. Local labels, by nature, are temporary; the .newblock directive resets them and terminates their scope.

A local label is a label in the form $n, where n is a single decimal digit. A local label, like other labels, points to an instruction word. Unlike other labels, local labels cannot be used in expressions. Local labels are not included in the symbol table.

After a local label has been defined and (perhaps) used, you should use the .newblock directive to reset it. The .text, .data, .ptext, and named sections also reset local labels. Local labels that are defined within an include file are not valid outside of the local file.

Example

This example declares the local label $1, resets it, and then declares it again.

```
 1                          ** Example of .newblock directive (PP)
 2
 3                                  .ref    ADDRA, ADDRB, ADDRC
 4          0000000000000076  BVAR    .set    76h
 5
 6 00000000 97871a4000000000! LABEL1: d7 = ADDRA
 7 00000008 fa839a4000000076          d3 = BVAR + d4
 8 00000010 f82af18000104100          d2 = d7 – d3
 9                                    br = [le]$1
10 00000018 9b821a4000000076          d2 = BVAR
11                                    br = $2
12
13
14 00000020 97841a4000000000! $1:     d4 = ADDRA
15 00000028 fa84ba4000000000! $2:     d4 = ADDRC + d5
16                                    .newblock
17                                    br = [lt]$1
18 00000030 fa84710000104100          d4 = d2 + d3
19 00000038 8800000000104100  $1:     NOP
```

Syntax

.option *option list*

Description

The **.option** directive selects several options for the assembler output listing. The option list is a list of options separated by commas; each option selects a listing feature. Valid options include:

B Limits the listing of .byte directives to one line.
H Limits the listing of .half directives to one line.
L Limits the listing of .long directives to one line.
T Limits the listing of .string directives to one line.
W Limits the listing of .word and .int directives to one line.
A Causes listing of all directives and data, and subsequent expansions, macros, and blocks.
D Controls the listing of certain directives (performs .drlist, see page CG:9-32).
M Turns off macro expansions in the listing.
N Turns off listing (performs .nolist).
O Turns on listing (performs .list).
R Resets the B, D, L, M, H, W, and T options.
X Produces a symbol cross-reference listing.

Options are not case-sensitive.

Example This example limits the listings of the .byte, .word, .long, and .string directives to one line each.

```
 1                                  ****************************************
 2                                  ** Limit the listing of .byte, .int,    **
 3                                  ** .long, and .string directives to 1   **
 4                                  ** line each   (PP)                     **
 5                                  ****************************************
 6                                          .option B, W, L, T
 7 00000000 00000000000000BD               .byte   –’C’, 0B0h, 5
 8 00000004 00000000AABBCCDD               .long   0AABBCCDDh, 536 + ’A’
 9 0000000c 00000000000015AA               .word   5546, 78h
10 00000014 0000000000000015               .int    010101b, 356q, 85
11 00000020 0000000000000045               .string ”Extended Registers”
12                                 ****************************************
13                                 **      Reset the listing options       **
14                                 ****************************************
15                                         .option R
16 00000032 00000000000000BD               .byte   –’C’, 0B0h, 5
   00000033 00000000000000B0
   00000034 0000000000000005
17 00000038 00000000AABBCCDD               .long   00AABBCCDDh, 536 + ’A’
   0000003c 0000000000000259
18 00000040 00000000000015AA               .word   5546, 78h
   00000044 0000000000000078
19 00000048 0000000000000015               .int    010101b, 356q, 85
   0000004c 00000000000000EE
   00000050 0000000000000055
20 00000054 0000000000000045               .string ”Extended Registers”
   00000055 0000000000000078
   00000056 0000000000000074
   00000057 0000000000000065
   00000058 000000000000006E
   00000059 0000000000000064
   0000005a 0000000000000065
   0000005b 0000000000000064
   0000005c 0000000000000020
   0000005d 0000000000000052
   0000005e 0000000000000065
   0000005f 0000000000000067
   00000060 0000000000000069
   00000061 0000000000000073
   00000062 0000000000000074
   00000063 0000000000000065
   00000064 0000000000000072
   00000065 0000000000000073
```

Syntax

```
.page
```

Description

The **.page** directive produces a page eject in the listing file. The source statement is not printed in the source listing, but the assembler increments the line counter when it encounters a new source statement. Using the .page directive to divide the source listing into logical divisions improves program readability.

Example

This example causes the assembler to begin a new page of the source listing.

Source file:

```
        .title   ”**** Page Directive Example ****”
;         .
;         .
;         .
        .page
```

Listing file:

```
MVP PP Macro Assembler     Version x.xx     Mon Nov 28 14:27:34 1994
 Copyright (c) 1993–1995    Texas Instruments Incorporated
***** Page Directive Example *****                                PAGE    1

       2                           ;              .
       3                           ;              .
       4                           ;              .
^L
MVP PP Macro Assembler     Version x.xx     Mon Nov 28 14:27:34 1994
 Copyright (c) 1993–1995    Texas Instruments Incorporated
***** Page Directive Example *****                                PAGE    2

 No Errors, No Warnings
```

Syntax

.sect ”*section name*”

Description

The **.sect** directive defines a named section that can be used like the default .text, .ptext, and .data sections. The assembler begins assembling source code into the named section.

The *section name* identifies a section that the assembler assembles code into. The name is significant to eight characters and must be enclosed in double quotes. Chapter 12, *Introduction to Common Object File Format*, provides additional information about named sections.

Example

This example defines two special-purpose sections, Sym_Defs and Vars, and assembles code into them:

```
 1                                  ********************************************
 2                                  ** Begin assembling into .ptext section (PP)**
 3                                  ********************************************
 4
 5 00000000                                 .ptext
 6 00000000 84445a40000000ff                d4 = d2 & 0x00ff; Assembled into .ptext
 8                                  ********************************************
 9                                  **  Begin assembling into Sym_Defs section  **
10                                  ********************************************
11
12 00000000                                 .sect   ”Sym_Defs”
13 00000000 000000003d4ccccd               .float  0.05     ; Assembled into Sym_Defs
14 00000004 00000000000000aa  X:           .word   0AAh     ; Assembled into Sym_Defs
15 00000008 84455a4000003456               d5 = d2 & 0x3456 ; Assembled into Sym_Defs
16
17                                  ********************************************
18                                  **   Begin assembling into Vars section      **
19                                  ********************************************
20
21 00000000                                 .sect   ”Vars”
22          0000001000000010  WORD_LEN      .set    16
23          0000002000000020  DWORD_LEN     .set    WORD_LEN * 2
24          0000000800000008  BYTE_LEN      .set    WORD_LEN / 2
25          0000005300000053  STR           .set    53h
26
27                                  ********************************************
28                                  **    Resume assembling into .ptext section    **
29                                  ********************************************
30
31 00000008                                 .ptext
32 00000008 888acf8000104100               d2 = 31 – d6     ; Assembled into .ptext
33 00000010 8a26c08540104100               d6 = [le] d6 + 1 ; Compare d6 to 31
34                                                          ; Increment if <= 31
35 00000018 0000000000000003               .byte   3, 4     ; Assembled into .ptext
   00000019 0000000000000004
36
37                                  ********************************************
38                                  **   Resume assembling into Vars section      **
39                                  ********************************************
40
41 00000000                                 .sect   ”Vars”
42 00000000 00000000000d0000               .field  13, WORD_LEN
43 00000000 00000000000d0a00               .field  0Ah, BYTE_LEN
44 00000000 00000000000d0a80               .field  10q, DWORD_LEN / 8
```

Syntax

symbol **.set** *value*
symbol **.equ** *value*

Description

The **.set** and **.equ** directives equate a constant value to a symbol. The symbol can then be used in place of a value in assembly source. This allows you to equate meaningful names with constants and other values. The two directives are equivalent.

symbol is a label that will be set to a value. It must appear in the label field.

value must be a well-defined expression; that is, all symbols in the expression must be previously defined in the current source module.

Undefined external symbols and symbols that are defined later in the module cannot be used in the expression. If the expression is relocatable, the symbol to which it is assigned is also relocatable.

The value of the expression appears in the object field of the listing. This value is not part of the actual object code and is not written to the output file.

Example

This example shows how symbols can be assigned with .set.

```
 1                                ********************************************
 2                                **   Equate symbol AUX_R1 to registers AR1  **
 3                                **   and use it instead of the register     **
 4                                ********************************************
 5          0000002300000023  AUX_R1  .set    d3
 6 00000000 609b1c0000104100           AUX_R1 = d2 * d7
 7
 8                                ********************************************
 9                                **  Set symbol index to an integer expr.    **
10                                **  and use it as an immediate operand      **
11                                ********************************************
12
13          0000003500000035  INDEX   .equ    100/2 + 3
14 00000008 fa859a4000000035          d5 =    INDEX + d4
15
16                                ********************************************
17                                **  Set symbol SYMTAB to a relocatable expr. **
18                                **  and use it as a relocatable operand     **
19                                ********************************************
20
21 00000010 000000000000000a  LABEL   .word   10
22          0000001100000011' SYMTAB  .equ    LABEL + 1
23
24                                ********************************************
25                                **   Set symbol NSYMS equal to the symbol    **
26                                **  INDEX and use it as you would INDEX    **
                                  ********************************************
27
28          0000003500000035  NSYMS   .set    INDEX
29 00000014 0000000000000035          .word   NSYMS
```

Syntax

.space *size in bytes*
.bes *size in bytes*

Description

The **.space** and **.bes** directives reserve *size* number of bytes in the current section and fill them with 0s. The section program counter is incremented to point to the word following the reserved space.

When you use a label with the .space directive, it points to the first byte reserved. When you use a label with the .bes directive, it points to the last byte reserved.

Example

This example reserves memory with the .space and .bes directives.

```
 1                          ********************************************
 2                          **   Begin assembling into .ptext section   **
 3                          ********************************************
 4 00000000                         .ptext
 5                          ********************************************
 6                          **  Reserve 0F0 bytes (15 words) in the      **
 7                          **              .ptext section              **
 8                          ********************************************
 9 00000000                         .space  0F0h
10 000000f0 0000000000000100        .word   100h, 200h
   000000f4 0000000000000200
11                          ********************************************
12                          **      Begin assembling into .data         **
13                          ********************************************
14 00000000                         .data
15 00000000 0000000000000049        .string "In .data"
   00000001 000000000000006E
   00000002 0000000000000020
   00000003 000000000000002E
   00000004 0000000000000064
   00000005 0000000000000061
   00000006 0000000000000074
   00000007 0000000000000061
16                          ********************************************
17                          **    Reserve 100 bytes in the .data section;  **
18                          **    Res_1 points to the first word that     **
19                          **    contains reserved bits                  **
20                          ********************************************
21 0000006b                 RES_1:  .bes    100
22 0000006c 000000000000000F       .word   15
23 00000070 000000000000006B"       .word   RES_1
24                          ********************************************
25                          **   Reserve 20 bytes in the .data section;    **
26                          **   Res_2 points to the last byte that       **
27                          **   contains reserved bits                   **
28                          ********************************************
29 00000087                 RES_2:  .bes    20
30 00000088 0000000000000036       .word   36h
31 0000008c 0000000000000087"       .word   RES_2
```

Syntax

```
.sslist
.ssnolist
```

Description

Two directives enable you to control substitution symbol expansion in the listing file:

- The **.sslist** directive allows substitution symbol expansion in the listing file. The expanded line appears below the actual source line.
- The **.ssnolist** directive inhibits substitution symbol expansion in the listing file.

By default, all substitution symbol expansion in the listing file is inhibited. Lines with the pound (#) character denote expanded substitution symbols.

Example

This example illustrates the default .ssnolist directive and the changes that occur in the listing when a .sslist is assembled.

```
      1                              ** sslist directive (PP)
      2
      3 00000000                             .bss    ADDRX, 1
      4 00000001                             .bss    ADDRY, 1
      5                              ADD2    .macro  ADDRA, ADDRB
      6                                      d7   =  ADDRA
      7                                      d7   =  d7 + ADDRB
      8                                      *ADDRB = d7
      9                                      .endm
     10
     11 00000000 888fcf8000104100            d7 = 31 – d6
     12 00000008                             ADD2    ADDRX, ADDRY
1       00000008 97871a4000000000–           d7   =  ADDRX
1       00000010 8a27fa4000000001–           d7   =  d7 + ADDRY
1       00000018 880000574f100001–           *ADDRY = d7
     13
     14                                      .sslist
     15
     16 00000020 8447da400000056a            d7 = d6 & 0x56a
     17 00000028 fa85f28e80104100            d5 = d5 + d7[n]d6
     18
     19 00000030                             ADD2    ADDRX, ADDRY
1       00000030 97871a4000000000–           d7   =  ADDRA
#                                            d7   =  ADDRX
1       00000038 8a27fa4000000001–           d7   =  d7 + ADDRB
#                                            d7   =  d7 + ADDRY
1       00000040 880000574f100001–           *ADDRB = d7
#                                            *ADDRY = d7
```

Syntax

.string ”*string*$_1$” [**,** ... **,** ”*string*$_n$”]

Description

The **.string** directive places 8-bit characters from a character string into the current section. Each string is either:

- An expression that the assembler evaluates and treats as a 8-bit signed number, or
- A character string enclosed in double quotes. Each character in a string represents a separate byte.

The assembler truncates any values that are greater than eight bits. You can have as many operands as fit on a single line (200 bytes).

If you use a label, it points to the location of the first byte that is initialized.

Note that when you use .string in a .struct/.endstruct sequence, .string defines a member’s size in number of characters; it does not initialize memory. For more information about .struct/.endstruct, see Section 9.8, *Symbol Directives*.

Example

This example places 8-bit values into words in the current section.

```
1                          ********************************************
2                          **        .string examples (PP)           **
3                          ********************************************
4
5  00000000 0000000000000041  Str_Ptr:        .string ”ABCD”
   00000001 0000000000000042
   00000002 0000000000000043
   00000003 0000000000000044
6  00000004 0000000000000041                  .string 41h, 42h, 43h, 44h
   00000005 0000000000000042
   00000006 0000000000000043
   00000007 0000000000000044
7  00000008 0000000000000041                  .string ”Austin”, ”Houston”
   00000009 0000000000000075
   0000000a 0000000000000073
   0000000b 0000000000000074
   0000000c 0000000000000069
   0000000d 000000000000006e
   0000000e 0000000000000048
   0000000f 000000000000006f
   00000010 0000000000000075
   00000011 0000000000000073
   00000012 0000000000000074
   00000013 000000000000006f
   00000014 000000000000006e
8  00000015 0000000000000030                  .string 36 + 12
```

Syntax

```
[ stag ]        .struct
[ mem0 ]  element       [expr0]
[ mem1 ]  element       [expr1]
          [.access]
     .        .          .
     .        .          .
[ memn ]  .tag   tagn   [ exprn]
     .        .          .
[ memN ]  element       [ exprN ]
[ size ]  .endstruct

label     .tag          stag
```

Description

The **.struct** directive assigns symbolic offsets to the elements of a data structure definition. This enables you to group similar data elements together and then let the assembler calculate the element offset. This is similar to a C structure or a Pascal record. The .struct directive does not allocate memory; it merely creates a symbolic template that can be used repeatedly.

The **.endstruct** directive terminates the structure definition.

On the PP only, when you use the **.access** directive, all structure member offsets are calculated relative to the .access directive. The .access directive may be placed between any two member definitions, or at the beginning or end of the structure. You may use only one .access directive per structure definition.

The **.tag** directive gives structure characteristics to a label, simplifying the symbolic representation and providing the ability to define structures that contain other structures. .tag does not allocate memory. The structure tag (stag) of a .tag directive must have been previously defined.

[*stag*] Is the structure's tag. Its value is associated with the beginning of the structure. If no stag is present, this tells the assembler to put the structure members in the global symbol table with their value being their absolute offset from the access point of the structure.

[$mem_{N/n}$] Is a label for a member of the structure. This label is absolute and equates to the present offset from the structure's point of reference.

element can be one of the following descriptors: .byte, .char, .double, .field, .float, .half, .int, .long, .short, .string, .ubyte, .uchar, .uhalf, .uint, .ulong, .ushort, .uword, or .word. An *element* can also be a complete declaration of a nested structure or union, or a structure or union declared by its tag. All of these are typical directives that initialize memory. Following a .struct directive, these directives describe the structure element's size. They *do not* allocate memory. A .tag directive is a special case because a stag must be specified (such as $stag_2$ in the definition).

[$expr_{N/n}$] Is an expression for the number of elements described. This value defaults to 1.

[tag_n] Is the nested structure's tag.

label Is a label for the total size of the structure.

Space is generally reserved for a structure in the .bss section by using the .bss directive. This directive takes two parameters, the symbol associated with the structure and the size. The size can be specified using the label of the structure, which gives its size in bytes. In this case the symbol is associated with a zero offset from the structure.

```
.bss  coord, xsiz ; memory allocation without
                  ; access declaration
```

On the PP only, the size can be specified by using the structure tag, in which case the size is equal to the size of the structure in bytes, and if the tag represents a structure with a nonzero access point, then the symbol is associated with the access point value of the structure.

```
.bss  coord, type ; memory allocation and access
                  ; declaration
```

Note: Directives That Can Appear in a struct/.endstruct Sequence

The only directives that can appear in a .struct/.endstruct sequence are element descriptors (explained above), conditional assembly directives, and the .align directive, which aligns the member offsets on word boundaries. Note that empty structures are illegal.

Note: .byte, .char, .ubyte, .uchar in a .struct/.endstruct Sequence

Note that when you use these directives in a .struct/.endstruct sequence, they define a member's size; they do not initialize memory. On the PP only, if you define a structure member as signed (.byte or .char) , then when it is loaded into a register sign will be extended. If a structure member is defined as unsigned (.ubyte or .uchar), then when it is loaded into a register it will be zero extended.

Example 1

```
 4                          real_rec    .struct      ; stag
 5          0000000000000000  nom         .word        ; member1 = 0
 6          0000000000000004  den         .word        ; member2 = 1
 7          0000000000000008  real_len    .endstruct   ; real_len = 4
 8
 9 00000000                             .bss real, real_len
10                                                    ; allocate mem rec
11                          real        .tag         real_rec
12 00000000 88000057df920001–           d7 =         *(xba + real.den)
13                                                    ; access structure element
```

Example 2

```
17
18                          cplx_rec    .struct
19          0000000000000000  reali       .tag real_rec  ; stag
20          0000000000000008  imagi       .tag real_rec  ; member1 = 0
21          0000000000000010  cplx_len    .endstruct     ; rec_len = 8
22
23                          complex     .tag  cplx_rec
24                                                    ; assign structure attrib
25 00000008                             .bss complex, cplx_len ; alloc space
26 00000008 88000056df920002–           d6 =  *(xba + complex.reali.nom)
27                                                    ; access structure element
28
29
```

Example 3

```
32
33                                    .struct         ; no stag puts mems into
34          0000000000000000  X       .word           ; global symbol table
35          0000000000000004  Y       .word           ; create 3 dim templates
36          0000000000000008  Z       .word
37          000000000000000c          .endstruct
```

Example 4

```
 1
 2
 3                          xstr    .struct         ; stag
 4          0000000000000000  m1      .word           ; offset = –12
 5          0000000000000004  m2      .uword          ; offset = –8
 6          0000000000000008  m3      .half           ; offset = –4
 7          000000000000000a  m4      .ubyte          ; offset = –2
 8          000000000000000b  m5      .byte           ; offset = –1
 9          000000000000000c  midx    .access         ; reference
10          000000000000000c  m6      .half           ; offset = 0
11          000000000000000e  m7      .uhalf          ; offset = 2
12          0000000000000010  m8      .word           ; offset = 4
13          0000000000000014  xsiz    .endstruct
14
15 00000000                           .bss    xyz, xsiz  ; reserve space for 1
16                                                       ; instance of structure
17 00000020                           .bss    xyz1, xstr ; reserve another
18                                                       ; instance and declare
19                                                       ; its access point
20
21                          xyz     .tag    xstr
22                          xyz1    .tag    xstr
23
24 00000000 88000052df800003–         a2 = &*(xba + xyz + midx) ;access point xyz
25
26 00000008 8800005ab5120002          d2 = *a2.xstr.m3          ;d2 = h * (a2–4)
27 00000010 88000052d5120001          d2 = *a2.xstr.m8          ;d2 = *(a2 + 4)
28
29 00000018 88000052df800008–         a2 = &*(xba + xyz1)       ;access point xyz1
30
31 00000020 8800005ab5120002          d2 = *a2.xstr.m3          ;d2 = h * (a2–4)
32 00000028 88000052d5120001          d2 = *a2.xstr.m8          ;d2 = *(a2 + 4)
```

Syntax

.system *symbol*$_1$ [**,** . . . **,** *symbol*$_n$]

Description

The **.system** directive identifies global symbols that can be referenced by both PP and MP modules that are linked together. It works similarly to the .global directive (CG:9-41) except that symbols declared with the .global directive are only accessible to other files assembled for the same processor.

Example

The following example shows that the MP and PP share the symbols y and z, regardless of which assembly language file they are defined in. Both processors define and reference their own versions of the symbol x. Figure 9–7 illustrates the PP and MP access to the various symbols.

File 1: pp1.asm

```
   .global x ; x is visible to PP, but not available to MP
   .system y ; make y visible to MP as well as PP

x: nop   ; definition of x local to PP
y: nop   ; definition of y available to both PP and MP
```

File 2: pp2.asm

```
   .global x ; get access to PP version of x
   .system y ; get access to system-wide y defined on PP
   .system z ; get access to system-wide z defined on MP

   .word x  ; reference PP version of x
   .word y  ; reference PP version of y
   .word z  ; reference MP version of z
```

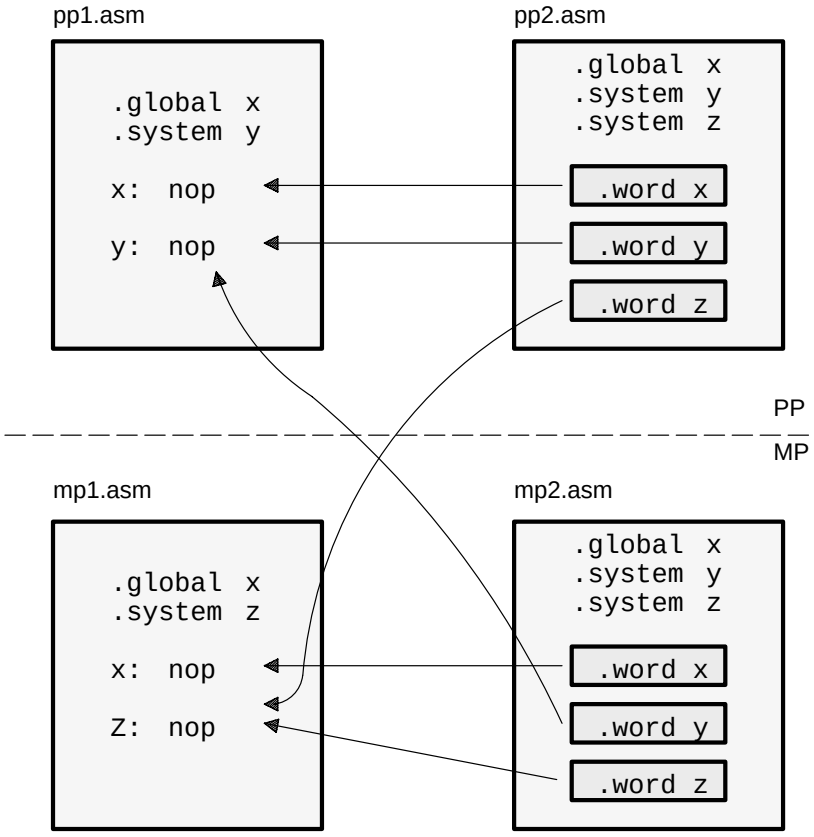
File 3: mp1.asm

```
   .global  x  ; x is visible to MP, but not
               ; available to PP
   .system  z  ; make z visible to PP as well as MP

x: nop   ; definition of x local to MP
z: nop   ; definition of z available to both MP and PP
```

File 4: mp2.asm

```
   .global  x  ; access MP version of x
   .system  y  ; access system-wide y defined on PP
   .system  z  ; access system-wide z defined on MP

   .word x  ; reference MP version of x
   .word y  ; reference PP version of y
   .word z  ; reference MP version of z
```

Figure 9–7. The .system Directive



Syntax

.tab *size*

Description

The **.tab** directive defines the tab size. Tabs encountered in the source input will be translated to size spaces in the listing. The default tab size is 8.

Example

Each of the following sets of lines consists of a single tab character followed by three NOP instructions, which illustrates the default tab size of 8 and the effect of the .tab directive with different operands:

```
 1 00000000   00020000          nop
 2 00000004   00020000          nop
 3 00000008   00020000          nop
 4                                        .tab 16
 5 0000000C   00020000                    nop
 6 00000010   00020000                    nop
 7 00000014   00020000                    nop
 8                          .tab 4
 9 00000018   00020000     nop
10 0000001C   00020000     nop
11 00000020   00020000     nop
```

Syntax

```
.text
.ptext
```

Description

The **.text** and **.ptext** directives are identical except that .text is used for MP sections and .ptext is used for PP sections. These directives tell the assembler to begin assembling into the default sections for MP and PP executable code, respectively. For more information on linking, see Chapter 13.

Source files must be completely dedicated to either MP or PP Assembly instructions. You may not mix MP and PP instructions in the same file.

Assigning PP assembly code to the .text section, or MP assembly code to the .ptext section, causes unpredictable results.

Note that .text or .ptext is the default section. Therefore, at the beginning of an assembly, the assembler assembles code into the .text or .ptext section unless you specify a different section directive (.data, .sect, etc.).

The SPC is set to 0 if nothing has yet been assembled into the section. If code has already been assembled into the section, the SPC is restored to its previous value in the section on reentry into the section.

Example 5 This example shows code assembled into the .text and .data sections (MP assembly). The .data section contains integer constants, and the .text section contains character strings and executable code.

```
 1
 2
 3                      ****************************************
 4                      **    Assemble into .data section (MP) **
 5                      ****************************************
 6
 7 00000000                      .data
 8 00000000   0A                 .byte   0Ah, 0Bh
   00000001   0B
 9
10                      ****************************************
11                      ** Begin assembling into .text section  **
12                      ****************************************
13
14 00000000                     .text
15 00000000   41       START:   .string "A","B","C"
   00000001   42
   00000002   43
16 00000003   58       END:     .string "X","Y","Z"
   00000004   59
   00000005   5A
17
18 00000008   4A3B0007          ADD     r7, r8, r9
19 0000000C   20FB0002          ADD     r2, r3, r4
20
21                      ****************************************
22                      **  Resume assembling into .data section **
23                      ****************************************
24
25 00000002                     .data
26 00000002   0C                .byte   0Ch, 0Dh
   00000003   0D
27
28                      ****************************************
29                      **  Resume assembling into .text section **
30                      ****************************************
31
32 00000010                     .text
33 00000010   51                .string "Quit"
   00000011   75
   00000012   69
   00000013   74
```

Example 6 This example shows code assembled into the .ptext and .data sections (PP assembly). The .data section contains integer constants, and the .text section contains character strings and executable code.

```
 1
 2
 3                          ****************************************
 4                          ** Begin assembling into .data section **
 5                          ****************************************
 6
 7 00000000                          .data
 8 00000000 000000000000000a         .byte   0Ah, 0Bh
   00000001 000000000000000b
 9
10                          ****************************************
11                          ** Begin assembling into .ptext section  **
12                          ****************************************
13
14 00000000                          .ptext
15 00000000 0000000000000041  START:  .string "A","B","C"
   00000001 0000000000000042
   00000002 0000000000000043
16 00000003 0000000000000058  END:    .string "X","Y","Z"
   00000004 0000000000000059
   00000005 000000000000005a
17
18 00000008 fa87b20000104100         d7 = d4 + d5
19 00000010 fa84710000104100         d4 = d2 + d3
20
21                          ****************************************
22                          **  Resume assembling into .data section **
23                          ****************************************
24
25 00000002                          .data
26 00000002 000000000000000c         .byte   0Ch, 0Dh
   00000003 000000000000000d
27
28                          ****************************************
29                          ** Resume assembling into .ptext section **
30                          ****************************************
31
32 00000018                          .ptext
33 00000018 0000000000000051         .string "Quit"
   00000019 0000000000000075
   0000001a 0000000000000069
   0000001b 0000000000000074
```

Syntax

.title ”*string*”

Description

The **.title** directive supplies a title that is printed in the heading on each listing page. The source statement itself is not printed, but the line counter is incremented. The *string* is a quote-enclosed title of up to 65 characters. If you supply more than 65 characters, the assembler truncates the string and issues a warning.

The assembler prints the title on the page that follows the directive, and on subsequent pages until another .title directive is processed. If you want a title on the first page, the first source statement must contain a .title directive.

Example

This example prints one title on the first page and a different title on succeeding pages.

Source file:

```
            .title   ”**** Title Directive Example ****”
;             .
;             .
;             .
            .title   ”**** Title Directive Example Part 2 ****”
```

Listing file:

```
MVP PP Macro Assembler     Version x.xx     Mon Nov 28 15:13:13 1994
 Copyright (c) 1993–1995    Texas Instruments Incorporated

**** Title Directive Example ****                                    PAGE    1

       2                            ;               .
       3                            ;               .
       4                            ;               .
^L
MVP PP Macro Assembler     Version x.xx     Mon Nov 28 15:13:13 1994
 Copyright (c) 1993–1995    Texas Instruments Incorporated

**** Title Directive Example Part 2 ****                             PAGE    2

 No Errors, No Warnings
```

Syntax

```
[ utag ]        .union      [ expr ]
[ umem0 ]element            [ expr0 ]
[ umem1 ]element            [ expr1 ]
      .          .              .
      .          .              .
[ umemn ].tag               tagn[,exprn]
      .          .              .
[ umemN ]element    [ exprN ]
[ size ]     .endunion

 label       .tag           utag
```

Description

The **.union** directive is a PP only directive that assigns symbolic offsets to the elements of alternate data structure definitions to be allocated in the same memory space. This enables you to define several alternate structures and then let the assembler do the element offset calculation. This is similar to a C union. The .union directive does not allocate any memory; it merely creates a symbolic template that can be used repeatedly.

A .struct definition may contain a .union definition, also .structs and .unions may be nested.

The **.tag** directive gives structure or union characteristics to a label, simplifying the symbolic representation and providing the ability to define structures or unions that contain other structures or unions. The .tag does not allocate memory. The structure or union tag of a .tag directive must have been previously defined.

The **.endunion** directive terminates the union definition.

[*utag*] Is the union's tag. Its value is associated with the beginning of the union. If no utag is present, this tells the assembler to put the union's members in the global symbol table. In this case each member must have a unique name.

[$umem_{N/n}$] Is a union member name, when the member is an element (defined below). A member does not allocate memory. It associates a size and type with the member name so that you can access different kinds of data from the same memory location.

element Is one of the following descriptors: .byte, .char, .double, .field, .float, .half, .int, .long, .short, .string, .ubyte, .uchar, .uhalf, .uint, .ulong, .ushort, .uword, or .word. An *element* can also be a complete declaration of a nested structure or union, or a structure or union declared by its tag. All of these are typical directives that initialize memory. Following a .union directive, these directives describe the union element's size. They *do not* allocate memory.

[$expr_{N/n}$] Is an expression for the number of elements described. This value defaults to 1.

[tag_n] Is a tag associated with this member, which must be a structure or union. Its value is associated with the beginning of the structure or union, and it allocates the space associated with the structure or union.

[*usize*] Is a label for the size of the largest union member. When a union is defined within a structure, the size attributed to the union is equivalent to the amount of space required to facilitate the union's largest member.

Note: Directives That Can Appear in a .union/.endunion Sequence

The only directives that can appear in a .union/.endunion sequence are element descriptors (described above), structure and union tags, conditional assembly directives, and the .align directive, which aligns the member offsets on word boundaries. Note that empty structures and unions are illegal.

Example This example shows unions with and without tags.

```
 1                              ******************************************
 2                              ***                   EXAMPLE 1  (PP)      ***
 3                              ******************************************
 4
 5                                      .global  employid
 6
 7                              xample  .union          ; utag
 8          0000000000000000    ival    .word           ; member1 = int
 9          0000000000000000    fval    .float          ; member2 = float
10                              sval   .string          ; member3 = string
11          0000000000000004    real_len .endunion      ; real_len = 2
12
13 00000000                             .bss employid, real_len
14                                                      ; allocate mem rec
15                              employid .tag xample    ; name an instance
16
17 00000000 880000575f920000-           d7 = *(xba + employid.fval)
18                                                      ; access union element
19
20                              ******************************************
21                              ***                   EXAMPLE 2   (PP)      ***
22                              ******************************************
23
24                                      .union          ; no utag puts mems into
25                                                      ; global symbol table
26          0000000000000000    x       .long           ; create 3 dim templates
27          0000000000000000    y       .float
28          0000000000000000    z       .word
29          0000000000000004    size_u  .endunion       ; size_u = 4
```

Syntax

symbol **.usect** **"***section name***",** *size in bytes [, alignment]*

Description

The .usect directive reserves space for variables in an uninitialized, named section. This directive is similar to the .bss directive; both simply reserve space for data and have no contents. However, .usect defines additional sections that can be placed anywhere in memory, independently of the .bss section.

symbol points to the first location reserved by this invocation of the .usect directive. The symbol corresponds to the name of the variable that you're reserving space for.

section name names the uninitialized section. The section name must be enclosed in double quotes; only the first 8 characters are significant.

size is a required parameter; it must be an absolute expression. The assembler allocates *size* bytes in the .usect section. There is no default size.

Size can also be specified with a structure tag name or a union tag name. In this case the size is equal to the size of the structure or union. If the tag represents a structure with a non–zero access point, then the symbol is associated with the access point value of the structure.

Other sections directives (.text, .ptext, .data, and .sect) end the current section and tell the assembler to begin assembling into another section. The .usect and the .bss directives, however, do not affect the current section. The assembler assembles the .usect and the .bss directives and then resumes assembling into the current section.

You can repeat the .usect directive to define more than one variable in the specified section. Variables that can be located contiguously in memory can be defined in the same section by using multiple .usect directives with the same section name.

For more information about COFF sections, see Chapter 12, *Introduction to Common Object File Format*.

Example

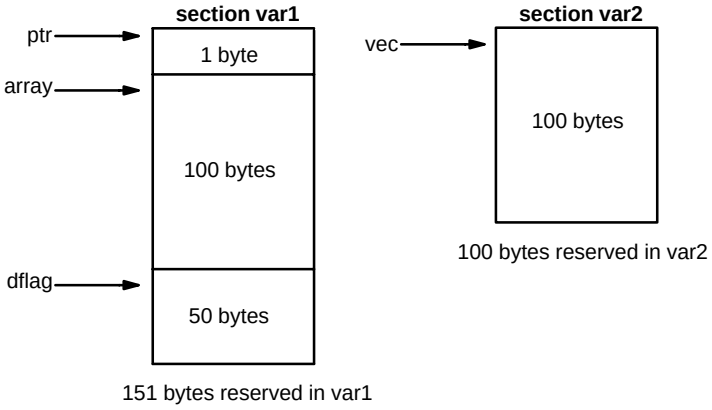
This example uses the .usect directive to define two uninitialized, named sections, var1 and var2. The symbol ptr points to the first byte reserved in the var1 section. The symbol array points to the first byte in a block of 100 words reserved in var1, and dflag points

to the first byte in a block of 50 bytes in var1. The symbol vec points to the first byte reserved in the var2 section.

Figure 9–8, shows how this example reserves space in two uninitialized sections, var1 and var2.

```
 1                              ************************************
 2                              *        Assemble into .ptext      *
 3                              ************************************
 4
 5 00000000                             .ptext
 6 00000000 fa87918000104100            d7 = d3 + d4
 7
 8                              ************************************
 9                              *     Reserve 1 byte in var1       *
10                              ************************************
11
12 00000000                    ptr     .usect  "var1", 1
13
14                              ************************************
15                              *   Reserve 100 more bytes in var1 *
16                              ************************************
17
18 00000001                    array   .usect  "var1", 100
19
20 00000008 8445f18000104100           d5 = d7 & d3 ; still in .ptext
21
22                              ************************************
23                              *  Reserve 50 more bytes in var1   *
24                              ************************************
25
26 00000065                    dflag   .usect  "var1", 50
27
28 00000010 8447fa4045678901           d7 = d7 & 0x45678901 ; in .ptext
29
30                              ************************************
31                              *      Reserve 100 bytes in var2   *
32                              ************************************
33
34 00000000                    vec     .usect  "var2", 100
35 00000018 97811bc009230000           br = 0x09230000 ; in .ptext
36                              ************************************
37                              * Declare an external .usect symbol *
38                              ************************************
39                                     .global array
```

Figure 9–8. The .usect Directive



Chapter 10

Macro Language

The assembler supports a macro language that enables you to create your own instructions. This is especially useful when a program executes a particular task several times. The macro language enables you to:

- Define your own macros and redefine existing macros
- Simplify long or complicated assembly code
- Access macro libraries created with the archiver
- Define conditional and repeatable blocks within a macro
- Manipulate strings within a macro
- Control expansion listing

Topics

10.1 Using Macros

Programs often contain routines that are executed several times. Instead of repeating the source statements for a routine, you can define the routine as a macro, then call the macro where you would normally repeat the routine. This simplifies and shortens your source program.

If you want to call a macro several times with different data each time, you can assign parameters to a macro. This enables you to pass different information to the macro each time you call it. The macro language supports a special symbol called a *substitution symbol*, which is used for macro parameters. In this chapter, we use the terms *macro parameters* and *substitution symbols* interchangeably.

Using a macro is a three-step process.

Step 1: Define the macro. You must define macros before you can use them in your program. Two methods can be used to define macros:

- Macros can be defined at the beginning of a **source file** or in a .include/.copy file. Refer to Section 10.2, *Defining Macros* for more information.
- Macros can also be defined in a **macro library**. A macro library is a collection of files in archive format created by the archiver. Each member of the archive file (macro library) may contain one macro definition corresponding to the member name. You can access a macro library by using the .mlib directive. Macro libraries are discussed in Section 10.4, *Macro Libraries*.

Step 2: Call the macro. After you have defined a macro, you can call it by using the macro name as an opcode in the source program; this process is referred to as a *macro call*.

Step 3: Expand the macro. The assembler expands your macros when the source program calls them. During expansion, the assembler passes arguments by variable to the macro parameters, replaces the macro call statement with the macro definition, then assembles the source code. By default, the macro expansions are printed in the listing file. You can turn off expansion listing by using the .mnolist directive. For more information, see Section 10.8, *Formatting the Output Listing*.

When the assembler encounters a macro definition, it places the macro name in the opcode table. This redefines any previously defined macro, library entry, directive, or instruction mnemonic that has the same name as the macro. This allows you to expand the functions of directives and instructions, as well as to add new instructions.

10.2 Defining Macros

You can define a macro anywhere in your program, but you must define the macro before you can use it. Macros can be defined at the beginning of a source file or in an .include/.copy file; they can also be defined in a macro library. For more information about macro libraries, see Section 10.4, *Macro Libraries*.

Macro definitions can be nested, and they can call other macros, but all elements of any macro must be defined in the same file. Nested macros are discussed in Section 10.9, *Using Recursive and Nested Macros*.

A macro definition is a series of source statements in the following format:

```
name    .macro [parameter1] [,parameter2]...[,parametern]
         model statements or macro directives
        [.mexit]
        .endm
```

name names the macro. You must place the name in the source statement's label field. Only the first 32 characters of a macro name are significant. The assembler places the macro name in the internal opcode table, replacing any instruction or previous macro definition with the same name.

.macro is a directive that identifies the source statement as the first line of a macro definition. You must place .macro in the opcode field.

parameters are optional substitution symbols that appear as operands for the .macro directive. Parameters are discussed in Section 10.3, *Macro Parameters/Substitution Symbols*.

model statements are instructions or assembler directives that are executed each time the macro is called.

macro directives are used to control macro expansion.

.mexit functions as a goto .endm. The .mexit directive is useful when error testing confirms that macro expansion will fail.

.endm terminates the macro definition.

Example 10–1 shows the definition, call, and expansion of a macro.

Example 10–1. Macro Definition, Call, and Expansion

Macro Definition: The following code defines a macro, add3, with 4 parameters:

```
1
2
3         *       add3
4         *
5         *       p3 = p1 + p2 + p3
6
7         add3    .macro  p1, p2, p3, addrp
8
9              d1 = p1
      10               d2 = p2
      11               d3 = p3
      12
      13               d4 = d1 + d2
      14               d5 = d3 + d4
      15               *addrp = d5
      16
      17
      18            .endm
```

Macro Call: The following code calls the add3 macro with 4 arguments:

```
   21             .global abc, def, ghi, adr
      22
      23 00000000 add3 abc, def, ghi, adr
```

Macro Expansion: The following code shows the substitution of the macro definition for the macro call. The assembler passes the arguments (supplied in the macro call) by variable to the parameters (substitution symbols).

```
1
1         00000000 97811a4000000000!         d1 = abc
1         00000008 97821a4000000000!         d2 = def
1         00000010 97831a4000000000!         d3 = ghi
1
1         00000018 fa84508000104100          d4 = d1 + d2
1         00000020 fa85918000104100          d5 = d3 + d4
1         00000028 880000555f100000!         *adr = d5
1
1
```

If you want to include comments with your macro definition, but *don't* want those comments to appear in the macro expansion, use an exclamation point to precede your comments. If you *do* want your comments to appear in the macro expansion, use an asterisk or semicolon. For more information about macro comments, see Section 10.7, *Producing Messages in Macros*.

10.3 Macro Parameters/Substitution Symbols

Macro parameters are symbols contained in the macro call that can be defined differently each time the macro is called. The macro language supports a special symbol, called a *substitution symbol*, which is used for macro parameters.

For example, in the call:

```
macro a,b,c
```

the parameters are a, b, and c. When you call the macro you replace a, b, and c with your values.

Substitution symbols

Substitution symbols are symbols that represent a character string. Besides being used as macro parameters, these symbols can also be used outside of macros to equate a character string to a symbol name.

Valid substitution symbols may be 32 characters long and *must begin with a letter*. The remainder of the symbol can be a combination of alphanumeric characters, underscores, and dollar signs.

Substitution symbols used as macro parameters are local to the macro they are defined in. You can define up to 32 local substitution symbols (including substitution symbols defined with the .var directive) per macro. For more information about the .var directive, see *Substitution symbols as local variables in macros,* page CG:10-15.

During macro expansion, the assembler passes arguments by variable to the macro parameters. The character-string equivalent of each argument is assigned to the corresponding parameter. Parameters without corresponding arguments are set to the null string. If the number of arguments exceeds the number of parameters, the last parameter is assigned the character-string equivalent of all remaining arguments.

If you pass a list of arguments to one parameter, or if you pass a comma or semicolon to a parameter, you must surround the arguments with quotation marks.

At assembly time, the assembler replaces the substitution symbol with its corresponding character string, then translates the source code into object code.

Example 10–2 shows the expansion of a macro with varying numbers of arguments.

Example 10–2. Calling a Macro With Varying Numbers of Arguments

Macro Definition

```
Parms  .macro a, b, c
   .string   ”:a: fish, :b: fish, :c: fish, new fish”
   .endm
```

Calling the Macro, Parms

```
   Parms  red, blue, old
   ; a = red
   ; b = blue
   ; c = old

   Parms  red, , old
   ; a = red
   ; b = ” ”
   ; c = old

   Parms  red, ”true blue”, old
   ; a = red
   ; b = true blue
   ; c = old

   Parms  red, blue, gnarly, old
   ; a = red
   ; b = blue
   ; c = gnarly, old

   Parms  ”green eggs, green ham, red”, blue, old
   ; a = green eggs, green ham, red
   ; b = blue
   ; c = old
```

Directives that define substitution symbols

You can manipulate substitution symbols with the **.asg** and **.eval** directives.

- The **.asg** directive assigns a character string to a substitution symbol.

  The syntax of the .asg directive is:

  .asg [**”**] *character string* [**”**]*, substitution symbol*

  The quotation marks are optional. If there are no quotation marks, the assembler reads characters up to the first comma and removes leading and trailing blanks. In either case, a character string is read and assigned to the substitution symbol.

Example 10–3 shows the same macro defined in Example 10–1, except that the .asg directive created a substitution symbol to represent a part of an operand.

Example 10–3. Using the .asg Directive

```
7                            add3    .macro  p1, p2, p3, ADDRP
8
9
      10                                       d1 = p1
      11                                       d2 = p2
      12                                       d3 = p3
      13                                       .asg    *, Ind
      14
      15                                       d4 = d1 + d2
      16                                       d5 = d3 + d4
      17
      18                                       Ind ADDRP = D5
      19
      20
      21                                       .endm
      22
      23
      24                                        .global abc, def, ghi, adr
      25
      26 00000000                               add3 4,5,2,0X2000
1
1
1        00000000 9781020000104100          d1 = 4
1        00000008 9782028000104100          d2 = 5
1        00000010 9783010000104100          d3 = 2
1                                           .asg    *, Ind
1
1        00000018 fa84508000104100          d4 = d1 + d2
1        00000020 fa85918000104100          d5 = d3 + d4
1
1        00000028 880000554f104562          * 0X2000 = d5
```

- The **.eval** directive performs arithmetic on numeric substitution symbols.

  The syntax of the .eval directive is

  .eval *well-defined expresssion, substitution symbol*

  The **.eval** directive evaluates the expression and assigns the *string value* of the result to the substitution symbol. If the expression is not well defined, the assembler generates an error and assigns the null string to the symbol.

Example 10–4 shows arithmetic performed on substitution symbols.

Example 10–4. Using the .eval Directive

```
.asg  1,counter
.loop 100
.word counter
.eval counter + 1,counter
.endloop
```

In Example 10–4, the .asg directive could be replaced with the .eval directive (.eval 1, counter) without changing the output. In simple cases like this, you can use .eval and .asg interchangeably. However, if you want to calculate a *value* from an expression, you must use the .eval directive.

The .asg directive only assigns a character string to a substitution symbol, while the .eval directive evaluates an expression and then assigns the character string equivalent to a substitution symbol.

Built-in substitution symbol functions

The following built-in substitution symbol functions enable you to make decisions based on the string value of substitution symbols. These functions always return a value, and they can be used in expressions. Built-in substitution symbol functions are especially useful in conditional assembly expressions. Parameters to these functions are substitution symbols or character-string constants.

In the following function definitions, *a* and *b* are parameters that represent substitution symbols or character string constants. The term *string*, used below, refers to the string value of the parameter.

Table 10–1. Functions and Their Return Values

Function	Return Value
$$symlen*(a)*	length of string *a*
$$symcmp*(a,b)*	< 0 if *a* < *b* 0 if *a* = *b* > 0 if *a* > *b*
$$firstch*(a,ch)*	index of the first occurrence of character constant *ch* in string *a*
$$lastch*(a,ch)*	index of the last occurrence of character constant *ch* in string *a*
$$isdefed*(a)*	1 if string *a* is defined in the symbol table 0 if string *a* is not defined in the symbol table
$$ismember*(a,b)*	top member of list *b* is assigned to string *a* 0 if *b* is a null string
$$iscons*(a)*	1 if string *a* is a binary constant 2 if string *a* is an octal constant 3 if string *a* is a hexadecimal constant 4 if string *a* is a character constant 5 if string *a* is a decimal constant
$$isname*(a*)	1 if string *a* is a valid symbol name 0 if string *a* is not a valid symbol name
$$isreg*(a)*	1 if string *a* is a valid predefined register name 0 if string *a* is not a valid predefined register name

For more information about substitution symbols, see Section 8.4, *Symbols*.

Example 10–5 shows built-in substitution symbol functions.

Example 10–5. Using Built-In Substitution Symbol Functions

```
       .asg   label, a                    ; a = label
       .if    ($$symcmp(a,”label”) = 0)   ; evaluates to true
        br = *(xba + calcintr)
       .endif
       .asg   ”x,y,z”   ,   list          ; list = x,y,z
       .if    ($$ismember(a, list))       ; a = x  list = y,z
        br = *(xba + prntlst)             ; sub      x
       .endif
```

Recursive substitution symbols

When the assembler encounters a substitution symbol, it attempts to substitute the corresponding character string. If that string is also a substitution symbol, the assembler performs substitution again. The assembler continues doing this until it encounters a token that is not a substitution symbol or until it encounters a substitution symbol that it has already encountered during this evaluation.

In Example 10–6, the x is substituted for z; z is substituted for y; and y is substituted for x. The assembler recognizes this as infinite recursion and ceases substitution.

Example 10–6. Recursive Substitution

```
        1
        2 00000000                    .bss    x, 4
        3 00000004                    .bss    y, 4
        4 00000008                    .bss    z, 4
        5
        6                             .asg    ″x″, z
        7                             .asg    ″z″, y
        8                             .asg    ″y″, x
        9
       10 00000000   00000000         .word   x
       11 00000004   00000004         .word   y
       12 00000008   00000008         .word   z
```

Forced substitution

In some cases, substitution symbols are not recognizable to the assembler. The forced substitution operator enables you to force the substitution of a symbol's character string. Simply surround a symbol with colons to force the substitution. Do not include any spaces between the colons and the symbol.

The syntax for the forced substitution operator is

:*symbol***:**

The assembler expands substitution symbols surrounded by colons before it expands other substitution symbols.

You can use the forced substitution operator only inside macros, and you cannot nest a forced substitution operator within another forced substitution operator.

Example 10–7 shows the use of the forced substitution operator.

Example 10–7. Using the Forced Substitution Operator

```
       1                              force   .macro x
       2                                      .asg  0, x
       3                                      .loop 4  ; .loop/.endloop
       4                              Aux:x:  .set    x
       5                                      .eval   x+1,x
       6                                      .endloop
       7                                      .endm
       8
```

The force macro would generate the following code:

```
       9 00000000                             force
1                                             .asg  0, x
1                                             .loop 4  ; .loop/.endloop
1                                     Aux:x:  .set    x
1                                             .eval   x+1,x
1                                             .endloop
2                 0000000000000000  Aux0  .set    0
2                                             .eval   0+1,x
2                 0000000000000001  Aux1  .set    1
2                                             .eval   1+1,x
2                 0000000000000002  Aux2  .set    2
2                                             .eval   2+1,x
2                 0000000000000003  Aux3  .set    3
2                                             .eval   3+1,x
      10
```

Accessing individual characters of subscripted substitution symbols

In a macro, you can access the individual characters (substrings) of a substitution symbol with subscripted substitution symbols. You must use the forced substitution operator for clarity.

You can access substrings in two ways.

- **:***symbol* **(***well-defined expression***):**

  This method of subscripting evaluates to a character string with one character.

- **:***symbol* **(***well-defined expression*$_1$ **,** *well-defined expression*$_2$**):**

  In this method, expression$_1$ represents the substring's starting position, and expression$_2$ represents the substring's length. You can specify exactly where to begin subscripting and the exact length of the resulting character string. *The index of substring characters begins with 1, not 0.*

Example 10–8 and Example 10–9 show built-in substitution symbol functions used with subscripted substitution symbols.

Example 10–8. Using Subscripted Substitution Symbols to Redefine an Instruction

```
vmscx  .macro    src1, src2, dst
   .var      tmp1, tmp2
   .asg      :dst(1):, tmp1
   .if    $$symcmp(tmp1, ”a”) == 0
   vmsc.ssd  src1, src1, dst, dst
   .else
   vmsc.ssd  src1, src2, 0, dst
   .endif
   .endm

   vmscx     r2, r4, a2
   vmscx     r2, r4, r6
```

In Example 10–8, subscripted substitution symbols redefine the add instruction so that it handles short immediates.

Example 10–9. Using Subscripted Substitution Symbols to Find Substrings

```
  substr    .macro    start,strg1,strg2,pos
            .var      len1,len2,i,tmp
            .if       $$symlen(start) = 0
            .eval     start,1
            .endif
            .eval     0,pos
            .eval     1,i
            .eval     $$symlen(strg1),len1
            .eval     $$symlen(strg2),len2
            .loop
            .break    i = (len2 – len1 + 1)
            .asg      ”:strg2(i,len1):”,tmp
            .if       $$symcmp(strg1,tmp) = 0
            .eval     i,pos
            .break
            .else
            .eval     i + 1,i
            .endif
            .endloop
            .endm

            .asg      0,pos
            .asg      ”ar1 ar2 ar3 ar4”,regs
            substr    1,”ar2”,regs,pos
            .word     pos
```

In Example 10–9, the subscripted substitution symbol is used to find a substring strg1 beginning at position start in the string strg2. The position of the substring strg1 is assigned to the substitution symbol pos.

Substitution symbols as local variables in macros

If you want to use substitution symbols as local variables within a macro, you can use the **.var** directive to define up to 32 local macro substitution symbols (including parameters) per macro. The .var directive creates temporary substitution symbols with the initial value of the null string. These symbols are not passed in as parameters, and after expansion they are lost.

.var sym_1 [,sym_2] ... [,sym_n]

The .var directive is used in Example 10–8 and Example 10–9.

10.4 Macro Libraries

One of the ways you can define macros is in a macro library. A macro library is a collection of files that contain macro definitions. You must use the archiver to collect these files, or members, into a single file (called an archive). Each member of a macro library contains one macro definition. The files in a macro library must be unassembled source files. The macro name and the member name must be the same, and the macro filename's extension must be .asm for the MP or .s for the PP. For example:

Macro Name	Filename in Macro Library
`simple`	`simple.asm`
`add3`	`add3.asm`

You can access the macro library by using the .mlib assembler directive.

.mlib *macro library filename*

When the assembler encounters an .mlib directive, it opens the library and creates a table of its contents. The assembler enters the names of the individual members within the library into the opcode tables as library entries; this redefines any existing opcodes or macros that have the same name. If one of these macros is called, the assembler extracts the entry from the library and loads it into the macro table. The assembler expands the library entry in the same manner as other macros. You can control the listing of library entry expansions with the .mlist directive. For more information about the .mlist directive, see Section 10.8, *Formatting the Output Listing*. Only macros that are actually called from the library are extracted, and they are extracted only once. For more information about the .mlib directive, see Section 9.6, *Directives That Reference Other Files*.

You can create a macro library with the archiver by simply including the desired files in an archive. A macro library is no different from any other archive, except that the assembler expects the macro library to contain macro definitions. The assembler expects **only** macro definitions in a macro library; putting object code or miscellaneous source files into the library may produce undesirable effects.

10.5 Using Conditional Assembly in Macros

The conditional assembly directives are **.if/.elseif/.else/.endif** and **.loop/.break/.endloop**. They can be nested within each other up to 32 levels deep. The format of a conditional block is:

.if *well-defined expression*

assemble code block when the expression is true (nonzero)

[.elseif *well-defined expression***]**

assemble code block when the .if expression is false (zero) and the .elseif expression is true (nonzero)

[.else *well-defined expression***]**

assemble code block when the .if expression and any .elseif expressions are false (zero)

.endif
terminate code block

The .elseif and .else directives are optional, and they can be used more than once within a conditional assembly code block. When .elseif and .else are omitted, and when the .if expression is false (zero), the assembler continues to the code following the .endif directive.

The **.loop/.break/.endloop** directives enable you to assemble a code block repeatedly. The format of a repeatable block is:

.loop [*well-defined expression*]

repeatedly assemble code block

[.break [*well-defined expression*]**]**

continue when the .break expression is false (zero); go to code following .endloop if .break expression is true (nonzero) or omitted

.endloop
assemble when .break directive is true (nonzero) or when .break expression is omitted and the loop count equals the expression.

The .loop directive's optional expression evaluates to the loop count. If the expression is omitted, the loop count defaults to 1024 unless the assembler encounters a .break directive.

The .break directive and its expression are optional. If the expression evaluates to false, the loop continues. The assembler breaks the loop when the .break expression evaluates to true or when the .break expression is omitted and the loop count equals *expression*. When the loop is broken, the assembler continues with the code after the .endloop directive.

Example 10–10, Example 10–11, and Example 10–12 show the .loop/.break/.endloop directives, properly nested conditional assembly directives, and built-in substitution symbol functions used in a conditional assembly code block.

Example 10–10. The .loop/.break/.endloop Directives

```
       .asg   1,x
       .loop            ;  ”infinite”

       .break (x == 10) ;  if x == 10, quit loop/break with
*                                                 expression

       .eval  x+1,x
       .endloop
```

Example 10–11. Nested Conditional Assembly Directives

```
       .asg   1,x
       .loop         ;  ”infinite”

       .if (x == 10);  if x == 10 quit loop
       .break        ;  force break
       .endif

       .eval   x+1,x
       .endloop
```

Example 10–12. Built-In Substitution Symbol Functions Used in a Conditional Assembly Code Block

```
    .fcnolist
*
*  branch depending on input
*
branch2  .macro  a

  .if $$symcmp (a, ”*”)
br = *(xba + starbranch)
  .elseif $$symcmp(a,”*+”)
br = *(xba + starplus)
  .elseif $$symcmp(a,”*–”)
br = *(xba + starminus)
  .else
br = *(xba + notdef)
  .endif
  .endm
```

For more information about conditional assembly directives, see Section 9.7, *Conditional Assembly Directives*.

10.6 Using Labels in Macros

All labels in an assembly language program must be unique. This includes labels in macros. If a macro is expanded more than once, its labels are defined more than once. *Defining a label more than once is illegal*. The macro language provides a method of defining labels in macros so that the labels are unique. Simply follow the label with a question mark, and the assembler will replace the question mark with a unique number. When the macro is expanded, *you will not see the unique number in the listing file*. Your label will appear with the question mark as it did in the macro definition. The syntax for a unique label is:

*label**?***

Example 10–13 shows unique label generation in a macro.

Example 10–13. Unique Labels in a Macro

```
1                           ; define macro
3                           min     .macro a, b, ans; find minimum
4
5                                     d2 = a
6                                     d7 = d2 – b
7                                     br = [le] m1?
8                                     d2 = b
9                                     br = m1?
      10
      11                            m1?  ans = *d2
      12
      13                                    .endm
      14
      15                            ;   call macro
      16
      17 00000000                           min 50, 100, 0x0756
1
1        00000000 9b821a4000000032          d2 = 50
1        00000008 f82f5a4000000064          d7 = d2 – 100
1        00000010 9b811bc500000028'         br = [le] m1?
1        00000018 9b821a4000000064          d2 = 100
1        00000020 9b811bc000000028'         br = m1?
1
1        00000028                  m1?  ans = *d2
      17
```

10.7 Producing Messages in Macros

The macro language supports three directives that enable you to define your own assembly-time error and warning messages. These directives are especially useful when you want to create messages specific to your needs. The last line of the listing file shows the error and warning counts. These counts alert you to problems in your code and are especially useful during debugging.

.emsg sends error messages to the listing file. The .emsg directive generates errors in the same way the assembler does, incrementing the error count and preventing the assembler from producing an object file.

.wmsg sends warning messages to the listing file. The .wmsg directive functions in the same manner as the .emsg directive does, but it increments the warning count.

.mmsg sends warnings or assembly-time messages to the listing file. The .mmsg directive functions in the same manner as the .emsg directive does but it does not set the error count.

Macro comments are comments that appear in the definition of the macro *but do not show up in the expansion of the macro*. An exclamation point in column 1 identifies a macro comment. If you want your comments to appear in the macro expansion, precede your comment with an asterisk or semicolon.

Example 10–14 shows user messages in macros.

Example 10–14. Producing Messages in a Macro

```
 2                      testparam   .macro x,y
 3
 4                          .if $$symlen(x) = 0
 5                              .emsg ”ERROR –– Missing parameter”
 6                              .mexit
 7                          .elseif $$symlen(y) = 0
 8                              .emsg ”ERROR –– Missing parameter”
 9                              .mexit
10                          .else
11                              ld x(r0),r1
12                              ld y(r0),r2
13                              add r1,r2,r3
14                          .endif
15                          .endm
16
17 00000000                 testparam 1,2
1
1                                   .if $$symlen(x) = 0
1                                       .emsg ”ERROR –– Missing parameter”
1                                       .mexit
1                                   .elseif $$symlen(y) = 0
1                                       .emsg ”ERROR –– Missing parameter”
1                                       .mexit
1                                   .else
1         00000000   08110001            ld 1(r0),r1
1         00000004   10110002            ld 2(r0),r2
1         00000008   18BB0001            add r1,r2,r3
1                                   .endif
18 0000000C                 testparam
1
1                                   .if $$symlen(x) = 0
1                                       .emsg ”ERROR –– Missing parameter”
 ********* USER ERROR  –  ERROR –– Missing parameter

 1 Error,  No Warnings
```

10.8 Formatting the Output Listing

Macros, substitution symbols, and conditional assembly directives may hide information. You may need to see this hidden information, so the macro language supports an expanded listing capability.

By default, the assembler shows macro expansions and false conditional blocks in the output list file. You may want to turn this listing off or on within your listing file. These sets of directives enable you to control the listing of this information:

❑ Macro and Loop Expansion Listing

.mlist expands macros and .loop/.endloop blocks. The**.mlist** directive prints to the listing all code encountered in those blocks. *By default, the assembler behaves as if you had used .mlist.*

.mnolist suppresses the listing expansion of macros and .loop/.endloop blocks.

❑ False Conditional Block Listing

.fclist causes the assembler to print to the listing file all false conditional blocks that do not generate code. Conditional blocks appear in the listing exactly as they appear in the source code. *By default, the assembler behaves as if you had used .fclist.*

.fcnolist suppresses the listing of false conditional blocks. Only the code in conditional blocks that actually assembles appears in the listing. The .if, .elsif, .else, and .endif directives do not appear in the listing.

❑ Substitution Symbol Expansion Listing

.sslist expands substitution symbols in the listing. This is useful for debugging the expansion of substitution symbols. The expanded line appears below the actual source line.

.ssnolist turns off substitution symbol expansion in the listing. *By default, the assembler behaves as if you had used .ssnolist.*

❑ Directive Listing

.drlist causes the assembler to print to the listing file all directive lines. *By default the assembler behaves as if you had used .drlist.*

.drnolist suppresses the printing of the following directives in the listing file:

.asg	.eval	.var	.sslist
.mlist	.fclist	.ssnolist	.mnolist
.fcnolist	.emsg	.wmsg	.mmsg
.length		.width	.break.

10.9 Using Recursive and Nested Macros

The macro language supports recursive and nested macro calls. This means that when you use nested macros, you can call other macros from your macro definition. You can nest macros up to 32 levels deep. When you use recursive macros, you call a macro from its own definition (the macro calls itself).

When you create recursive or nested macros, you should pay close attention to the arguments that you pass to macro parameters, because the assembler uses dynamic scoping for parameters. This means that the called macro uses the environment of the macro from which it was called.

Example 10–15 illustrates the use of nested macros. Notice that the macro definitions do not produce object code, but the call in the last statement generates object code for the entire nested macro.

Example 10–15. Using Nested Macros

Source File:

```
   .list
mac1   .macro op1,op2
mac2      .macro op3,op4
          .word op3
          .word op4
       .endm
   mac2 op1,op2
   .endm

   mac1 9,10
```

Listing File:

```
        3                       mac1    .macro  op1,op2
        4                       mac2        .macro op3,op4
        5                                   .word op3
        6                                   .word op4
        7                               .endm
        8                           mac2 op1,op2
        9                           .endm
       10
       11 00000000                  mac1 9,10
1                               mac2        .macro op3,op4
1                                           .word op3
1                                           .word op4
1                                       .endm
1          00000000                 mac2 op1,op2
2          00000000   00000009                .word 9
2          00000004   0000000A                .word 10
```

Example 10–16 illustrates the use of recursive macros. The fact macro produces assembly code necessary to calculate the factorial of n, where n is an immediate value. The fact macro stores that value at data memory address loc. The fact macro accomplishes this by calling fact1, which calls itself recursively.

Example 10–16. Using Recursive Macros

```
    .global addr, fact1

fact1 .macro
           .if   n > 1    ; multiply present factorial
                          ; by present position
               d6 = *(xba + loc)
               d3 = d7 * d6
               *(xba + loc) = d3 ; save result
               .eval  n–1, n
               call = fact1
           .endif
       .endm

fact   .macro n, loc ; n is an integer constant
                       ; loc memory address – n!
                       ; 0! = 1! = 1
      .if     n < 2
               d7 = 1
               *(xba + loc) = d7
           .else
               d7 = n
               *(xba + loc) = d7
               .eval n–1, n
               call = fact1
           .endif
       .endm

   fact 5, addr
```

10.10 Macro Directives Summary

Table 10–2. Creating Macros

Mnemonic and Syntax	Description
macname **.macro** [*parameter*$_1$]...[,*parameter*$_n$]	Define macro
.mlib *filename*	Identify library containing macro definitions
.mexit	Go to .endm
.endm	End macro definition

Table 10–3. Manipulating Substitution Symbols

Mnemonic and Syntax	Description
.asg [“]*character string*[”], *substitution symbol*	Assign character string to substitution symbol
.eval *well-defined expression*, *substitution symbol*	Perform arithmetic on numeric substitution symbols
.var *substitution symbol*[$_1$...,*substitution symbol*$_n$]	Define local macro symbols

Table 10–4. Conditional Assembly

Mnemonic and Syntax	Description
.if *well-defined expression*	Begin conditional assembly
.elseif *well-defined expression*	Optional conditional assembly block
.else *well-defined expression*	Optional conditional assembly block
.endif	End conditional assembly
.loop [*well-defined expression*]	Begin repeatable block assembly
.break [*well-defined expression*]	Optional repeatable block assembly
.endloop	End repeatable block assembly

Table 10–5. Producing Assembly-Time Messages

Mnemonic and Syntax	Description
.emsg	Send error message to standard output
.wmsg	Send warning message to standard output
.mmsg	Send assembly-time message to standard output

Table 10–6. Formatting the Listing

Mnemonic and Syntax	Description
.drlist	Enable listing of all directive lines (default)
.drnolist	Inhibit listing of the following directives lines: .asg, .eval, .var, .sslist, .mlist, .fclist, .ssnolist, .mnolist, .fcnolist, .emsg, .mmsg, .wmsg, .length, .width, and .break
.fclist	Allow false conditional code block listing (default)
.fcnolist	Inhibit false conditional code block listing
.mlist	Allow macro listings (default)
.mnolist	Inhibit macro listings
.sslist	Allow expanded substitution symbol listing
.ssnolist	Inhibit expanded substitution symbol listing (default)

Chapter 11

Assembler Error Messages

The assembler issues several types of error messages:

- Warnings
- Errors
- User-defined messages

When the assembler has completed its second pass, it reports any errors that it encountered during the assembly. It also prints these errors in the listing file (if one is created); an error is printed following the source line that incurred it.

This chapter discusses the three types of assembler error messages; they are listed by classes. Most errors are fatal; if an error is not fatal or if it is a macro error, this is noted in the list. The error messages are divided into *classes* which have similar descriptions and similar corrective actions. Each error message class has a *Description* of the problem and an *Action* that suggests possible remedies. Each error message consists of its class number and text showing the specific error that has been detected.

Topics

11.1 Warning Messages Produced by the Assembler

W0000: operand warnings

Bad use of g modifier, ignored

Invalid condition specified

Invalid field index — ignored

No operands expected. Operands ignored

Specified alignment is outside accessible memory — ignored

Status not affected by multiplies, status protection ignored

Trailing operands ignored

Unrecognized protection bit specified

Description The assembler has encountered operands that it did not expect or that are not in proper form. The assembler will ignore these operands.

Action This is a warning message. Check your source to determine what caused the problem and whether you need to correct the source.

W0001: truncated value warnings

Field value truncated to *value*

Field width truncated to *size in bits*

Line too long, will be truncated

Power of 2 required, *next larger power of 2* assumed

Section Name is limited to 8 characters

String is too long – will be truncated

Value truncated

Value truncated to byte size

Description The assembler has encountered values longer than it can accept in the given field. The assembler truncates the value.

Action Check your source to make sure the result will be acceptable, or to change the source if an error has occurred.

W0002: arithmetic expression warnings

Address expression will wrap-around

Expression will overflow, value truncated

Description The assembler has performed a calculation that will produce the indicated result, which may or may not be acceptable.

Action Make sure the result will be acceptable, or change the source if an error has occurred.

W0003: symbolic debug directive usage warning

.sym for function name required before .func

Description This warning informs you of problems with symbolic debugger directives. The specifics are obvious from the text.

Action Make sure the result will be acceptable, or change the source if an error has occurred.

W0004: mnemonic or directive misuse

Access point has already been defined

Access only allowed in top-most structure definition

Open block(s) at EOF

Description A directive or mnemonic has been encountered where it is semantically illegal or where it would have no effect. The directive or mnemonic is ignored by the assembler.

Action Re-evaluate the use of this directive or mnemonic to see if it needs to be moved to another place where it would be more effective.

11.2 Error Messages Generated by the Assembler

There are several classes of assembler errors. The error messages are grouped numerically by class. The classes are:

- **E0000—E0099** Syntax errors, see subsection 11.2.1
- **E0100—E0199** Label definition errors, see subsection 11.2.2
- **E0200—E0299** Expression errors, see subsection 11.2.3
- **E0300—E0399** Symbol errors, see subsection 11.2.4
- **E0400—E0499** Symbol table errors, see subsection 11.2.5
- **E0500—E0599** Macro errors, see subsection 11.2.6
- **E0600—E0699** Macro library errors, see subsection 11.2.7
- **E0700—E0799** Symbolic debugging directive errors, see subsection 11.2.8
- **E0800—E0899** Instruction errors, see subsection 11.2.9
- **E0900—E0999** Directive errors, see subsection 11.2.10
- **E1000—E1099** File I/O errors, see subsection 11.2.11
- **E1100—E1199** Pipeline conflict errors, see subsection 11.2.12
- **E1200—E1299** Processor resource allocation errors, see subsection 11.2.13
- **E1300—E1399** Assembler limit errors, see subsection 11.2.14

11.2.1 Syntax Errors

E0000: general syntax errors

Comma required to separate arguments

Comma required to separate parameters

Left parenthesis expected

Matching right parenthesis is missing

Missing right quote of string constant

Right parenthesis expected

Syntax Error

Description Required syntax is not present.

Action Correct the source as indicated.

E0001: invalid symbol names

Illegal symbol name

Description An illegal symbol name has been detected.

Action Correct the source as indicated.

E0002: invalid mnemonics

Invalid directive specification

Invalid mnemonic specification

Description The instruction, macro, or directive specified was not recognized.

Action Correct the source as indicated.

E0003: invalid operands

Cluttered character operand encountered

Cluttered string constant operand encountered

Cluttered identifier operand encountered

Illegal condition operand or combination

Illegal indirect memaddr specification

Invalid binary constant specified

Incorrect bit symbol for specified status register

Invalid constant specification

Invalid decimal constant specified

Invalid float constant specified

Invalid hex constant specified

Invalid immediate expression or shift value

Invalid octal constant specified

Invalid operand *x*

Shift value out of range

Description The instruction, parameter, or other operand specified was not recognized.

Action Correct the source as indicated.

E0004: illegal operands

Absolute, well-defined integer value expected

Address expression required

Address offset is too big

Data size must be equal to pointer size

Data unit operand is not valid for any ALU port

Expecting dual memory addressing

Identifier operand expected

Illegal character argument specified

Illegal endian flag specified

Illegal input to barrel rotator

Illegal operand to scale operator

Illegal register for register indirect addressing mode

Illegal string constant operand specified

Invalid addressing mode

Invalid identifier, *sym*, specified

Invalid macro parameter specified

No parameters available for macro arguments

Not expecting indirect operand

Not expecting immediate value

Pointer too big for this data size

Single character operand expected

src2 and dst2 operands for divi instructions must be D registers

String constant or substitution symbol expected

Structure/Union tag symbol expected

Substitution symbol operand expected

Description The instruction, parameter, or other operand specified was not legal for this syntax.

Action Correct the source as indicated.

E0005: missing operands

Missing field value operand

Missing operand(s)

Description A required operand is missing.

Action Correct the source as indicated.

E0006: unmatched conditional assembly directives

.break must occur within a loop

Conditional assembly mismatch

Matching .macro missing

No matching .endif specified

No matching .endloop specified

No matching .if specified

No matching .loop specified

Open block(s) inside macro

Unmatched .endloop directive

Unmatched .if directive

Description The assembler has encountered a directive that requires a matching directive, but it could not find the matching directive.

Action Correct the source as indicated.

E0007: conditional assembly nesting problems

Conditional nesting is too deep

Loop count out of range

Description These errors are associated with conditional assembly loops.

Action Check documentation to determine limits and see why they were exceeded.

E0008: unmatched structure / union definition directives

Bad use of .access directive

Matching .union directive is not present

Matching .struct directive is not present

Description In a structure or union definition, the assembler has encountered a directive that requires a matching directive, but it could not find the matching directive.

Action Correct the source as indicated.

E0009: illegal use of an operator

Cannot apply bitwise NOT to floats

Illegal assignment operator

Illegal bitwise operation in address unit instruction

Illegal use of ++ or –– operator, ++/–– must be applied to an address register

Illegal struct/union reference dot operator

Illegal use of multiplicative operator

Illegal use of unary minus operator

Scale operator only valid in address expressions

Structure or union tag symbol not found

Unary operator must be applied to a constant

Description The operator specified was not legal for the given operands.

Action Correct the source as indicated.

11.2.2 Label Definition Errors

E0100: label required

Label missing

.setsym requires a label

Description The given directive requires a label, but none is specified.

Action Correct the source as indicated.

E0101: label not permitted

Labels are not allowed with this directive

Standalone labels not permitted in structure/union defs

Description The given directive does not permit a label, but one is specified.

Action Correct the source as indicated.

E0102: illegal use of local labels

Local label *label number* is multiply defined

Local label *label number* is not defined in this section

Local labels can’t be used with directives

Description The given label was used illegally.

Action Correct the source as indicated.

11.2.3 Expression Errors

E0200: general expression errors

Binary operator can’t be applied

Cannot combine symbols from different sections

Can’t subtract a symbol from a constant

Difference between segment symbols not permitted

Division by zero is illegal

Expression must be absolute integer value

Offset expression must be integer value

Operation cannot be performed on given operands

Unary operator can’t be applied

Value of expression has changed due to jump expansion

Well-defined expression required

Description There is an Illegal operand combination, or an arithmetic type is required but not present.

Action Correct the source as indicated.

E0201: floating-point expression problems

Absolute operands required for FP operations!

Cannot apply bitwise NOT to floats

Floating-point divide by zero

Floating-point overflow

Floating-point underflow

Floating-point expression required

Illegal floating-point expression

Invalid floating-point operation

Description A specific problem associated with floating-point expressions is present.

Action Correct the source as indicated.

11.2.4 Symbol Errors

E0300: general symbol errors

Cannot equate an external symbol to an external symbol

Cannot redefine this section name

Cannot tag an undefined symbol

Empty structure or union definition

Illegal structure or union tag

Redefinition of *sym* attempted

Structure / union member previously defined

Structure / union member, *sym*, not found

Structure tag can't be global

Symbol can't be defined in terms of itself

Symbol expected in label field

Symbol, *sym*, has already been defined

Symbol, *sym* is not defined in this source file

Symbol, *sym* is operand to both .ref and .def

Description Usually these errors indicate that there was an attempt to redefine a symbol or attempts to define a symbol illegally.

Action Correct the source as indicated.

E0301: general substitution symbol errors

Cannot redefine local substitution symbol

Substitution stack overflow

Substitution symbol not found

Description Usually these errors indicate that there was an attempt to redefine a substitution symbol or attempts to define a substitution symbol illegally.

Action Correct the source as indicated.

11.2.5 Symbol Table Errors

E0400: general symbol table errors

Symbol table entry is not balanced

Description These errors usually indicate that a symbolic debugging directive does not have a complementing directive (that is, the code contains a .block without an .endblock).

Action Correct the source as indicated.

11.2.6 Macro Errors

E0500: general macro errors

Macro argument string is too long

Missing macro name

Too many variables declared in macro

Description These errors are usually generated when a macro definition has been corrupted in some way.

Action Correct the source as indicated.

E0501: macro definition directive match problems

Macro definition not terminated with .endm

Matching .endm missing

Matching .macro missing

.mexit directive outside macro definition

No active macro definition

Description These errors usually indicate that a macro directive does not have a complementing directive (that is, there is a .macro without a .endm).

Action Correct the source as indicated.

11.2.7 Macro Library Errors

E0600: macro library accessing errors

Bad archive entry for *macro name*

Bad archive name

Can’t read a line from archive entry

***library name* macro library not found**

***library name* is not in archive format**

Description Usually these errors indicate problems reading or writing to a macro library archive file. It is likely that the creation of the archive file was not done properly.

Action Correct the source as indicated.

11.2.8 Symbolic Debugging Directive Errors

E0700: illegal use of symbolic debug directive

Illegal structure/union member

No structure/union currently open

.sym not allowed inside structure/union

Description These errors are generated when a symbolic debug directive is used in an inappropriate location.

Action Correct the source as indicated.

11.2.9 Instruction Errors

E0800: general instruction errors

Can’t resolve d0 code’s multiply bits

Condition code disagreement

Expecting parallel instruction

Explicit parallel EALU required

Illegal combination of six-operand operations

Illegal data-unit operations specified

Illegal use of parallel operator

Instructions not permitted in structure/union definitions

Invalid negation of EALU’s function code

Missing required EALU operation

Non-zero dms permitted only with split or round multiples

Status protection code disagreement

Too many operands, can only map operands to 3 ports

Description These errors are normally target specific.

Action Correct the source as indicated.

11.2.10 Directive Errors

E0900: illegal use of directive

Cannot change version after 1st instruction

Can’t include a file inside a loop or macro

Illegal structure member

Illegal structure definition contents

Illegal union member

Illegal union definition contents

Invalid load-time label

Invalid structure/union contents

.setsect only valid if absolute listing produced (use –a option)

.setsym only valid if absolute listing produced (use –a option)

.var allowed only within macro definitions

Description These errors indicate cases where specific directives are not permitted, because they will cause a corruption of the object file. Many directives are not permitted inside of structure/union definitions.

Action Correct the source as indicated.

11.2.11 File I/O Errors

E1000: unable to read, write, or find a file

Include/copy file not found or opened

Description These errors are normally self-explanatory.

Action Correct the source as indicated.

11.2.12 Pipeline Conflict Errors

E1100: register latency detected in straight-line code

Pipeline conflict detected

Description These errors indicate a register latency detection in straight-line code. Register latencies can be detected for those processors that have fairly deep pipelines, but only in straight-line code.

Action Correct the source as indicated.

11.2.13 Processor Resource Allocation Errors

E1200: processor resources overextended

Condition code bits oversubscribed

Processor resource allocation conflict

Description These errors are detected in cases where several instructions in parallel cannot be executed at the same time, given the limits of the processor's resources. These errors will only be detected for those processors that are highly parallel.

Action Correct the source as indicated.

11.2.14 Assembler Limit Errors

E1300: general assembler limits exceeded

Copy limit has been reached

Exceeded limit for macro arguments

Macro nesting limit exceeded

Description These errors indicate that some limit of the assembler has been reached.

Action Consult documentation to check for limits, and determine how they have been exceeded.

Chapter 12

Introduction to Common Object File Format

The assemblers and linker create object files that can be executed by a TMS320C8x device. The format that these object files are in is called common object file format, or COFF.

COFF makes modular programming easier, because it encourages you to think in terms of blocks of code and data when you write an assembly language program. These blocks are known as sections. Both the assembler and the linker provide directives that allow you to create and manipulate sections.

This chapter provides an overview of COFF sections and includes the following topics:

Topics

12.1 Sections

The smallest unit of an object file is called a **section**. A section is a block of code or data that will ultimately occupy contiguous space in the TMS320C8x memory map. Each section of an object file is separate and distinct from the other sections. COFF object files always contain four default sections:

.ptext usually contains PP executable code.

.text usually contains MP executable code.

.data usually contains initialized data.

.bss usually reserves space for uninitialized variables.

The .pbss section will also be default section in COFF files generated from PP code. In addition, the assembler and linker allow you to create, name, and link *named* sections that are used in the same way as the .data, .text, .ptext, .bss, and .pbss sections.

It is important to note that there are two basic types of sections:

Initialized contain data or code. The .text, .ptext, and .data sections are initialized; named sections created with the .sect assembler directive are also initialized.

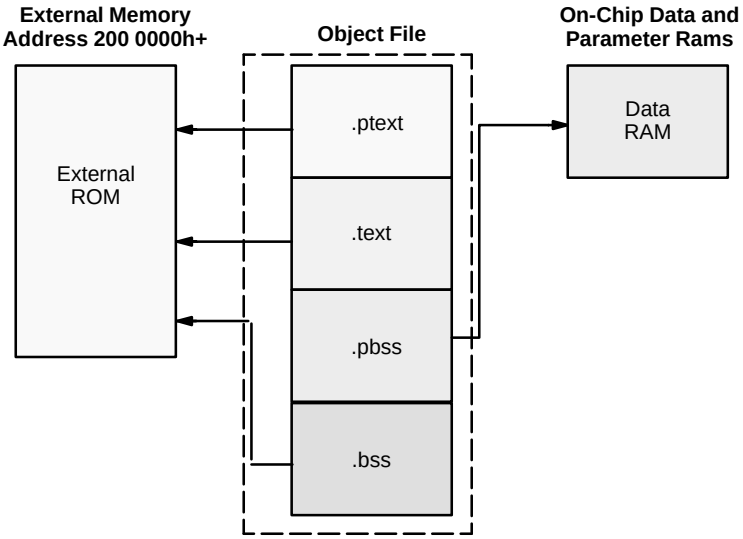
Uninitialized reserve space in the memory map for uninitialized data. The .bss section is uninitialized; named sections created with the .usect assembler directive (such as .pbss) are also uninitialized.

The assembler provides several directives that allow you to associate various portions of code and data with the appropriate sections. The assembler builds these sections during the assembly process, creating an object file that is organized like the object file shown in Figure 12–1.

One of the linker's functions is to relocate sections into the target memory map; this is called *allocation*. Because most systems contain several different types of memory, using sections can help you to use target memory more efficiently. All sections are independently relocatable; you can place sections into various blocks of target memory. For example, you can define a section that contains an initialization routine and then allocate the routine into a portion of the memory map that contains EPROM.

Figure 12–1 shows the relationship between sections in an object file and a hypothetical target memory.

Figure 12–1. Partitioning Memory Into Logical Blocks



12.2 How the Assembler Handles Sections

The assembler's main function with regard to sections is to identify the portions of an assembly language program that belong in a particular section. The assembler has six directives that support this function:

- **.bss**
- **.usect**
- **.text**
- **.ptext**
- **.data**
- **.sect**

The .bss and .usect directives create *uninitialized sections*; the .text, .ptext, .data, and .sect directives create *initialized* sections.

Note: Default Section Directive

If you don't use any of the sections directives, the assembler assembles everything into the .text section for the MP and the .ptext section for the PP.

12.2.1 Uninitialized Sections

Uninitialized sections reserve space in TMS320C8x memory; they are usually allocated into on-chip RAM for the PP and extended memory for the MP. These sections have no actual contents in the object file; they simply reserve space in memory. A program can use this space at runtime for creating and storing variables.

Uninitialized data areas are built by using the .bss and .usect assembler directives. The .bss directive reserves space in the .bss section. The .usect directive reserves space in a specific uninitialized named section. For example, the .pbss section is created using the .usect directive. Each time you invoke the .bss directive, the assembler reserves more space in the .bss section. Each time you invoke the .usect directive, the assembler reserves more space in the specified named section.

The syntax for these directives is:

	.bss *symbol***,** *size in bytes* [**,** *alignment*]
symbol	**.usect "***section name***",** *size in bytes* [**,** *alignment*]

symbol points to the first byte reserved by this invocation of the .bss or .usect directive. The symbol corresponds to the name of the variable that you're reserving space for. It can be referenced by any other section and can also be declared as a global symbol (with the .global or .sysmem assembler directives).

size in bytes is an absolute expression. The .bss directive reserves *size* bytes in the .bss section; the .usect directive reserves *size* bytes in *section name*.

For the PP only, size can also be specified with a structure tag name or union tag name. In this case the *size* is equal to the size of the structure/union represented by the tag. If the tag represents a structure with a non–zero access point, then the symbol is associated with the access point value of the structure.

section name tells the assembler which named section to reserve space in. For more information about named sections, see subsection 12.2.3, *Named Sections*.

alignment is an optional parameter that tells the assembler to align to the specified byte. *Alignment* may equal any power of 2.

The .ptext, .text, .data, and .sect directives tell the assembler to stop assembling into the current section and begin assembling into the indicated section. The .bss and .usect directives, however, *do not* end the current section and begin a new one; they simply escape from the current section temporarily. The .bss and .usect directives can appear anywhere in an initialized section without affecting the contents of the initialized section.

12.2.2 Initialized Sections

Initialized sections contain executable code or initialized data. The contents of these sections are stored in the object file and placed in TMS320C8x memory when the program is loaded, usually in the external memory. Each initialized section is separately relocatable and may reference symbols that are defined in other sections. The linker automatically resolves these section-relative references.

Four directives tell the assembler to place code or data into an initialized section. The syntax for these directives is:

```
.ptext
.text
.data
.sect  "section name"
```

When the assembler encounters one of these directives, it stops assembling into the current section (the directive acts as an implied end-current-section command). It then assembles subsequent code into the directive-related section until it encounters another .text, .data, .sect, or .ptext directive. (Remember, this is unlike the result of .bss of .usect directives which do not function as "end current section" commands. See Section 12.2.3, *Named Sections*.)

Remember that .text is the executable code section for the MP, while .ptext is the executable code section for the PP. Each source file must be completely dedicated to either MP or PP assembly code. Files may not mix these two. If MP code is assembled into the .ptext section, or PP code is assembled into the .text section, the results are unpredictable.

Sections are built up through an iterative process. For example, when the assembler *first* encounters a .data directive, the .data section is empty. The statements following this first .data directive are assembled into the .data section (until the assembler encounters a .text, .sect, or .ptext directive). If the assembler encounters subsequent .data directives, it *adds* the statements following these .data directives to the statements that are already in the .data section. This creates a single .data section that can be allocated contiguously into memory.

12.2.3 Named Sections

Named sections are sections that *you* create. You can use them like the default .ptext, .text, .data, and .bss sections, but they are assembled separately from the default sections.

For example, repeated use of the .text directive builds up a single .text section in the object file. When linked, this .text section is allocated into memory as a single unit. Suppose there is a portion of executable code (perhaps an initialization routine) that you don't want allocated with .text. If you assemble this segment of code into a named section, it will be assembled separately from .text, and you will be able to allocate it into memory separately from .text. Note that you can also assemble initialized data that is separate from the .data section, and you can reserve space for uninitialized variables that is separate from the .bss section.

Two directives let you create named sections:

- The **.usect** directive creates sections that are used like the .bss section. These sections reserve space in RAM for variables.
- The **.sect** directive creates sections, like the default .text and .data sections, that can contain code or data. These directives create named sections with relocatable addresses.

The syntax for these directives is:

symbol **.usect** "*section name", size* [**,** *alignment*]
.sect *"section name"*

The *section name* parameter is the name of the section. Section names are significant to eight characters. You can create up to 32,767 separate named sections. The optional alignment parameter tells the assembler to align to the specified byte.

Each time you invoke one of these directives with a new name, you create a new named section. Each time you invoke one of these directives with a name that was already used, the assembler assembles code or data (or reserves space) into the section with that name. *You cannot use the same names with different directives*. That is, you cannot create a section with the .usect directive and then try to use the same section with .sect.

12.2.4 Section Program Counters

The assembler maintains a separate program counter for *each* section. These program counters are known as *section program counters*, or *SPC*s.

An SPC represents the current address within a section of code or data. Initially, the assembler sets each SPC to 0. As the assembler fills a section with code or data, it increments the appropriate SPC. If you *resume* assembling into a section, the assembler remembers the appropriate SPC's previous value and continues incrementing the SPC at that point.

The assembler treats each section as if it began at address 0; the linker relocates each section according to its final location in the memory map.

12.2.5 An Example That Uses Sections Directives

Example 12–1 shows how you can build COFF sections incrementally, using the sections directives to swap back and forth between the different sections. You can use sections directives to:

- Begin assembling into a section for the first time
- Continue assembling into a section that already contains code. In this case, the assembler simply appends the new code to the code that is already in the section.

The format in Example 12–1 is a listing file. Example 12–1 shows how the SPCs are modified during assembly. A line in a listing file has four fields: Field 1 contains the source code line counter, Field 2 contains the section program counter, Field 3 contains the object code, and Field 4 contains the original source statement.

Example 12–1. Using Sections Directives (MP)

```
 1
 2                      ********************************************
 3                      **  Assemble an initialized table into .data **
 4                      ********************************************
 5 00000000                     .data
 6 00000000   00000011 coeff    .word          011h,022h,033h
   00000004   00000022
   00000008   00000033
 7                      ********************************************
 8                      **   Reserve space in .bss for a variable    **
 9                      ********************************************
10 00000000                     .bss           buffer,10
11                      ********************************************
12                      **              Still in .data              **
13                      ********************************************
14 0000000C   00000123 ptr      .word          0123h
15                      ********************************************
16                      **    Assemble code into the .text section   **
17                      ********************************************
18 00000000                     .text
19 00000000   40345000 add:     LD             0x2000184(r0),r8
   00000004   02000184
20 00000008   4A2D0001 aloop:   SUB            1,r8,r9
21 0000000C   02647FFF          BBZ            aloop,r9,31
22                      ********************************************

23                      **   Another initialized table into .data    **
24                      ********************************************
25 00000010                     .data
26 00000010   000000AA ivals    .word          0AAh, 0BBh, 0CCh
   00000014   000000BB
   00000018   000000CC
27                      ********************************************
28                      ** Define another section for more variables **
29                      ********************************************
30 00000000             var2    .usect         "newvars", 1
31 00000001             inbuf   .usect         "newvars", 7
32                      ********************************************
33                      **       Assemble more code into .text       **
34                      ********************************************
35 00000010                     .text
36 00000010   4A11000A mpy:     LD             0Ah(r8),r9
37 00000014   426C0123 mloop:   ADD            0x123,r9,r8
38 00000018   E2247FFF          BBZ            mloop,r8,3
39                      ********************************************
40                      **  Define a named section for int. vectors **
41                      ********************************************
42 00000000                     .sect          "vectors"
43 00000000   00000011          .word          011h, 033h
   00000004   00000033
```

Field 1 Field 2 Field 3 Field 4

As Figure 12–2 shows, the file in Example 12–1 creates five sections:

.text contains 28 bytes (1Ch bytes or seven 32-bit words) of object code. The SPC counts the bytes in hex notation.

.data contains 28 bytes (1Ch bytes or seven 32-bit words) of object code.

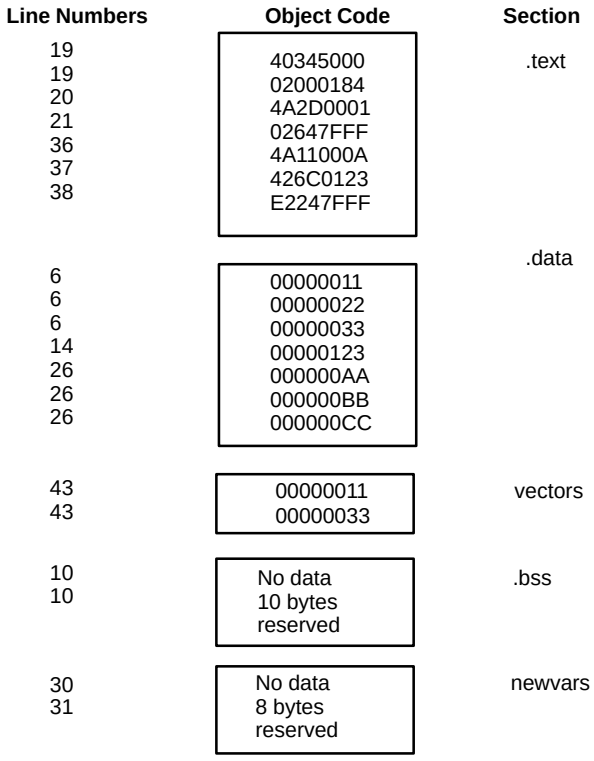
vectors is a named section created with the .sect directive; it contains two 32-bit words (eight bytes) of initialized data.

.bss reserves ten bytes in memory.

newvars is a named section created with the .usect directive; it reserves eight bytes in memory.

The second column shows the object code that is assembled into these sections; the first column shows the source statements that generated the object code.

Figure 12–2. Object Code Generated by Example 12–1 (MP)



12.3 How the Linker Handles Sections

The linker has two main functions with regard to sections. First, the linker uses the sections in COFF object files as building blocks; it combines input sections (when more than one file is being linked) to create output sections in an executable COFF output module. Second, the linker chooses memory addresses for the output sections.

The linker provides two directives that support these functions:

- The **MEMORY directive** allows you to define the memory map of a target system. You can name portions of memory and specify their starting addresses and their lengths.
- The **SECTIONS directive** tells the linker how to combine input sections and where to place the output sections in memory.

It is not always necessary to use linker directives. If you don't use them, the linker uses the target processor's default allocation algorithm described in subsection 12.3.1, *Default Memory Allocation*. When you *do* use linker directives, you must specify them in a linker command file.

Refer to the following sections for more information about linker command files and linker directives:

Section	
13.4	Linker Command Files
13.6	The MEMORY Directive
13.7	The SECTIONS Directive
13.10	Default Allocation Algorithm

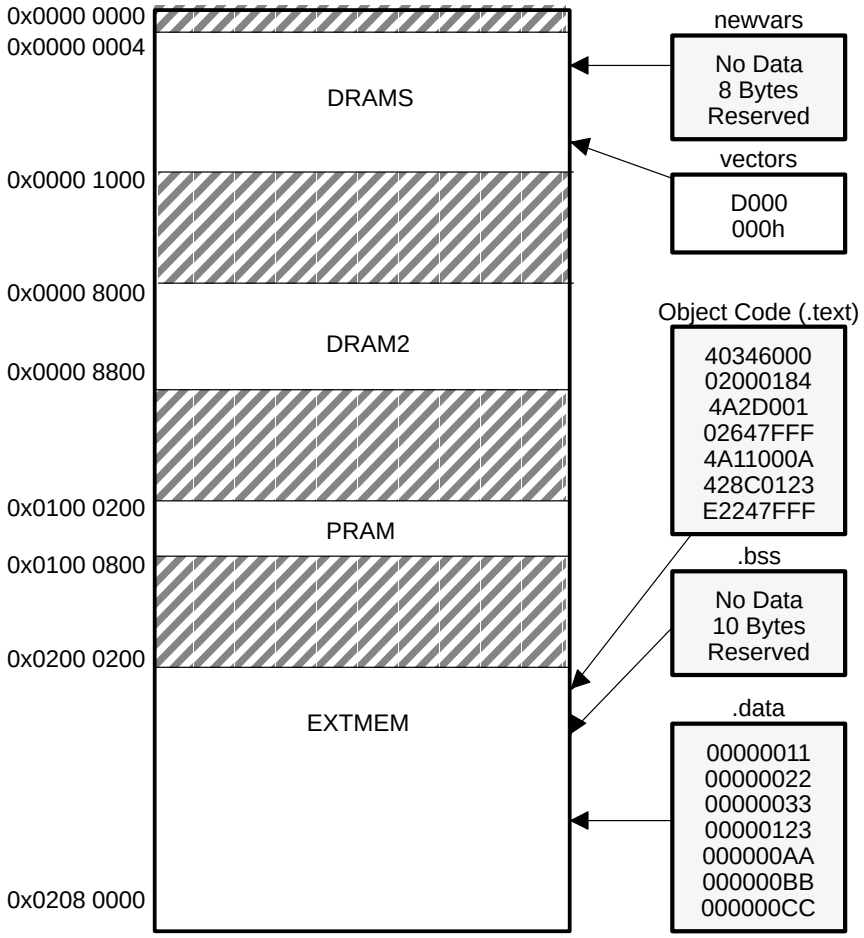
12.3.1 Default Memory Allocation

You can link files without specifying a MEMORY or SECTIONS directive. The linker uses a default model to combine sections (if necessary) and allocate them into memory. When using the default model, the linker:

1) Allocates .text into external memory
2) Allocates .ptext into external memory
3) Allocates .bss into external memory
4) Allocates .data into external memory
5) Allocates .pbss into data RAM (PP)
6) If you are linking C code with the –c, –cr, or –pc options, the linker will allocate additional sections by default. For more information about these additional sections, see Section 13.10, *Default Allocation Algorithm*.
7) Allocates all remaining sections into available configured memory using a best-fit algorithm. For more information about this algorithm, see Section 13.10, *Default Allocation Algorithm*.

Figure 12–3 shows how a single file would be allocated into memory using default allocation for the TMS320C8x. Note that the linker does not actually place object code into memory; it assigns addresses to sections so that a loader can place the code in memory.

Figure 12–3. Default Allocation for the Object Code in Figure 12–2 (MP)



As Figure 12–3 shows, the linker:

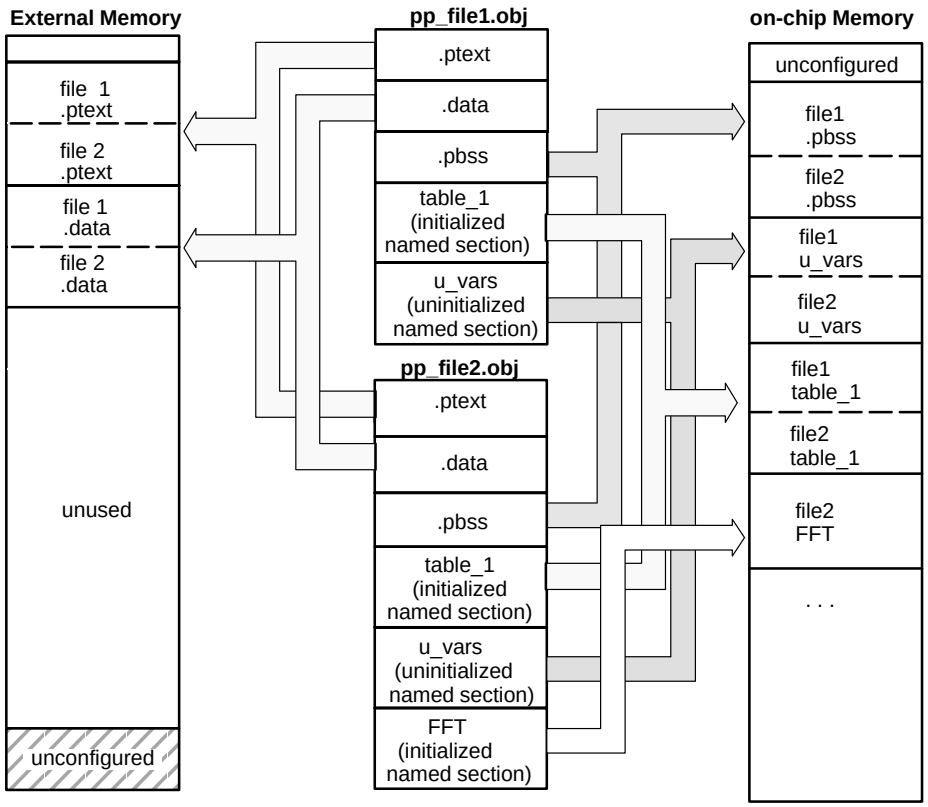
1) Allocates the .text section first into external RAM beginning at 02000000h. The text section contains 28 bytes (01Ch or seven 32-bit words) of object code.
2) Allocates the .bss section (the .ptext section is empty) beginning at 0200001Ch. The .bss section reserves 10 bytes in memory.
3) Allocates the .data section (aligned to a 4-byte boundary) beginning at 02000028h.
4) Allocates the named section newvars into the PP data RAM beginning at 00000004h. The newvars section contains no data, but reserves 8 bytes of memory.
5) Allocates the named section vectors into the PP data RAM (aligned to a 4-byte boundary) beginning at 0000000Ch. The initialized section vectors contains 8 bytes of data.

Note that there are reserved spaces throughout the on-chip address range 00h–1FFF FFFFh. There are specific sections of

memory set aside for data and parameter RAMs and instruction caches for the MP and each PP.

Figure 12–4 shows a simplified illustration of how two files would be linked together. When you link several files by using the default algorithm, the linker combines all input sections that have the same name into one output section that has this same name. For example, the linker combines the .ptext sections from the two input files to create one .ptext output section.

Figure 12–4. Combining Input Sections From Two Files (Default Allocation, PP)



In Figure 12–4, pp_file1.obj and pp_file2.obj each contain .ptext, .data, and .pbss sections; an initialized named section called table_1; and an uninitialized named section called u_vars. file2.obj also contains an initialized named section called FFT. As Figure 12–4 shows, the linker:

1) Combines pp_file1.obj .ptext with pp_file2.obj .ptext to form one .ptext output section. The .ptext output section is allocated at address 0200 0000h in external memory.

2) Combines pp_file1.obj .data with pp_file2.obj .data to form the .data output section. The .data output section is allocated into external memory following the .ptext output section.

3) Combines pp_file1.obj .pbss with pp_file2.obj .pbss to form the .pbss output section. The .pbss output section is allocated at address 0004h in on-chip memory.

4) All remaining sections are placed into the remaining available on-chip memory, then available external memory

For more information about default allocation algorithms, see Section 13.10, *Default Allocation Algorithm*.

12.3.2 Placing Sections in the Memory Map

Figure 12–3 and Figure 12–4 illustrate the linker's default methods for combining sections and allocating them into memory. Sometimes you may not want to use the default setup. For example, you may not want to combine all of the .data sections into a single .data output section. Or, you might want to place a named section instead of the .text section at memory address 0x20000000. Most memory maps are composed of various types of memories (DRAM, ROM, EPROM, etc.) in varying amounts; you may want to place a section in a particular type of memory.

The following illustrations show another possible combination of the sections from Figure 12–3:

Example 12–2	defines a sample TMS320C8x memory map using the linker's MEMORY and SECTIONS directives.
Figure 12–5	shows how the ranges defined in Example 12–2 fit into the TMS320C8x memory map.

Example 12–2. TMS320C80 MEMORY and SECTIONS Directives

```
MEMORY
{   /* Data RAM sections */
   BSS:  origin = 0x0800,  length = 100h
    /* Program sections */
   VECS:  origin = 0x04000000,  length = 020h
   CODE:  origin = 0x04000050,  length = 0200h
    /* Data sections    */
   RAMB2: origin = 0x04001000,  length = 020h
   RAMB0: origin = 0x04001050,  length = 100h
   RAMB1: origin = 0x04001150,  length = 100h
}

SECTIONS
{
   vectors:              >  VECS
   .text:                >  CODE
   .data:                >  RAMB2
   .bss:                 >  BSS
   newvars:              >  RAMB1
}
```

- The MEMORY directive in Example 12–2 defines five memory ranges:

VECS	CODE
RAMB2	RAMB0
BSS	RAMB1

  This MEMORY definition defines the following ranges:

  - 0x0000 0800 — 0x0000 08FF in on-chip DATA RAM
  - 0x0400 0000 — 0x0400 001F in external memory
  - 0x0400 0050 — 0x0400 024F in external memory
  - 0x0400 1000 — 0x0400 101F in external memory
  - 0x0400 1050 — 0x0400 114F in external memory
  - 0x0400 1150 — 0x0400 124F in external memory

  All undefined ranges are *unconfigured*. As far as the linker is concerned, these areas do not exist, and no code or data can be loaded into them. Whenever you use the MEMORY directive, only the memory ranges that the directive defines can contain code or data.

- The SECTIONS directive in Example 12–2 defines the order in which the sections are allocated into memory. The vectors section is allocated into VECS in external memory; .text is allocated into the CODE range in external memory; the .data section is allocated into the RAMB2 range in external memory; .bss is allocated into BSS in data RAM; newvars is allocated into RAMB1 in external memory.

Figure 12–5. Memory Map Defined by Example 12–2

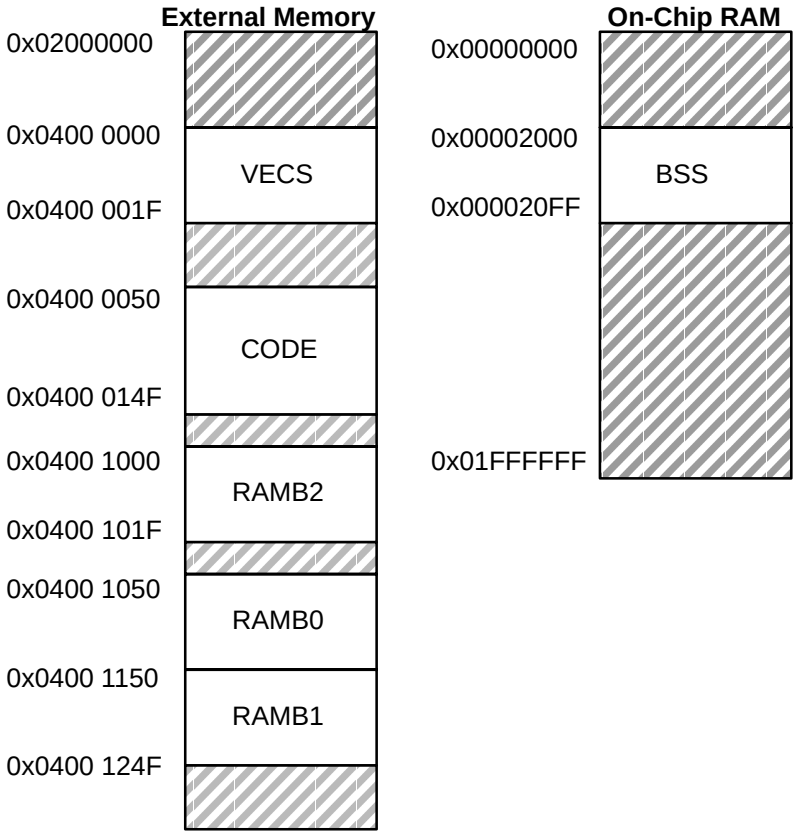


Figure 12–5 shows how the ranges defined by the MEMORY directive in Example 12–2 fit into the memory map.

12.4 Relocation

The assembler treats each section as if it begins at address 0. All relocatable symbols (labels) are relative to address 0 in their sections. Of course, all sections can't actually begin at address 0 in memory, so the linker relocates sections by:

- Allocating sections into the memory map so that they begin at the appropriate address
- Adjusting symbol values to correspond to the new section addresses
- Adjusting references to relocated symbols to reflect the adjusted symbol values

The linker uses *relocation entries* to adjust references to symbol values. The assembler creates a relocation entry each time a relocatable symbol is referenced. The linker then uses these entries to patch the references after the symbols are relocated. Example 12–3 contains a code segment for the TMS320C8x PP that generates relocation entries.

Example 12–3. Code That Generates Relocation Entries

```
       1                                    .ref x
       2 00000000                           .ptext
       3 00000000 97801BC000000000!         call = x
       4 00000008 97811A4000000018'         d1   = y
       5 00000010 8800000000104100          nop
       6 00000018                     y:
```

In Example 12–3, both symbols X and Y are relocatable. Y is defined in the .ptext section of this module; X is defined in another module. When assembled, X has a value of 0 (the assembler assumes all undefined external symbols have values of 0), and Y has a value of 0x18 (relative to address 0 in the .ptext section). The assembler generates two relocation entries, one for X and one for Y. The reference to X is an external reference (indicated by the ! character in the listing). The reference to Y is to an internally defined relocatable symbol (indicated by the ' character in the listing).

After the code is linked, suppose that X is relocated to address 02000000h. Suppose also that the .ptext section is relocated to begin at address 02000200h. Y now has a relocated value of 02000218h. The linker uses the two relocation entries to patch the two references in the object code:

```
97801BC000000000      call = x    becomes  97801BC002000000      call = x
97811A4000000018      d1   = y    becomes  97811A4002000018      d1   = y
```

Each section in a COFF object file has a table of relocation entries. The table contains one relocation entry for each relocatable reference in the section. The linker usually removes relocation entries after it uses them. This prevents the output file from being relocated again (when it is loaded or relinked or). A file that contains no relocation entries is an **absolute file** (all its addresses are absolute addresses). If you want the linker to retain relocation entries, invoke the linker with the –r option.

12.5 Runtime Relocation

It may be necessary or desirable at times to load code into one area of memory and run it in another. For example, you may have performance-critical code in a ROM-based system. The code must be loaded into ROM, but would run faster in RAM.

The linker provides a simple way to specify this. In the SECTIONS directive, you can optionally direct the linker to allocate a section twice: once to set its load address, and again to set its run address. Use the *load* keyword for the load address and the *run* keyword for the run address.

The load address determines where a loader will place the raw data for the section. Any references to the section (such as labels in it) refer to its run address. The application must copy the section from its load address to its run address; this does not happen automatically just because you specify a separate run address.

If you provide only one allocation (either load or run) for a section, the section is allocated only once and will load and run at the same address. If you provide both allocations, the section is actually allocated as if it were two different sections of the same size.

Uninitialized sections (such as .bss or .pbss) are not loaded, so the only significant address is the run address. The linker allocates uninitialized sections only once: if you specify both run and load addresses, the linker warns you and ignores the load address.

For a complete description of runtime relocation, see Section 13.8, *Specifying a Section's Runtime Address*.

12.6 Loading a Program

The linker produces executable COFF object modules. An executable object file has the same COFF format as object files that are used as linker input; the sections in an executable object file, however, are combined and relocated to fit into target memory.

In order to run a program, the data in the executable object module must be transferred, or *loaded*, into target system memory.

Several methods can be used for loading a program, depending on the execution environment. Some of the more common situations are listed below.

- The TMS320C8x debugging tools, including the software simulator, and XDS emulator have built-in loaders. Each of these tools has a LOAD command that invokes a loader; the loader reads the executable file and copies the program into target memory.
- If you are using a ROM- or EPROM-based system, you can use the hex conversion utility, which is shipped as part of the TMS320C8x software development toolset, to convert the executable COFF object module into one of several object file formats. You can then use the converted file with an EPROM programmer to burn the program into an EPROM.
- Some TMS320C8x programs are loaded under the control of an operating system or monitor software running directly on the target system. In this type of application, the target system usually has an interface to the file system on which the executable module is stored. You must write a custom loader for this type of system. The loader must comprehend the file system to access the file and the memory organization of the target system to load the program into memory.

12.7 Symbols in a COFF File

A COFF file contains a symbol table that stores information about symbols in the program. The linker uses this table when it performs relocation. Debugging tools can also use the symbol table to provide symbolic debugging.

12.7.1 External Symbols

External symbols are symbols that are defined in one module and referenced in another module. You can use the .global or .sysmem assembler directive to identify symbols as external. Alternatively, an external symbol can be either *defined* (DEF) or *referenced* (REF).

Defined (DEF)	Defined in the current module and used in another module
Referenced (REF)	Referenced in the current module, but defined in another module

The following code segment illustrates these definitions.

```
x:          ;  Define x
       CMP       y ,r8,r6          ;  Reference y
       .def   x                    ;  DEF of x
       .ref   y                    ;  REF of y
```

The .def definition of x says that it is an external symbol defined in this module and that other modules can reference x. The .ref definition of y says that it is an undefined symbol that is defined in another module.

The assembler places both x and y in the object file's symbol table. When the file is linked with other object files, the entry for x defines unresolved references to x from other files. The entry for y causes the linker to look through the symbol tables of other files for y's definition.

The linker must match all references with corresponding definitions. If the linker cannot find a symbol's definition, it prints an error message about the unresolved reference. This type of error prevents the linker from creating an executable object module.

12.7.2 The Symbol Table

The assembler always generates an entry in the symbol table when it encounters an external symbol (both definitions and references). The assembler also creates special symbols that point to the beginning of each section; the linker uses these symbols to relocate references to other symbols in a section.

The assembler does not usually create symbol table entries for any other type of symbol because the linker does not use them. For example, labels are not included in the symbol table unless they are declared with .global, .def, .ref, or .sysmem. For symbolic debugging purposes, it is sometimes useful to have entries in the symbol table for each symbol in a program. To accomplish this, invoke the assembler with the –s option.

Chapter 13

Linker Description

This chapter describes the operation of the TMS320C8x linker. Chapter 15, *Linker Error Messages* describes the error messages produced by the linker. Chapter 14, *Linking PP and MP Files: An Extended Example*, contains a TMS320C8x specific compiling and linking example.

Topics

13.1 Linker Development Flow

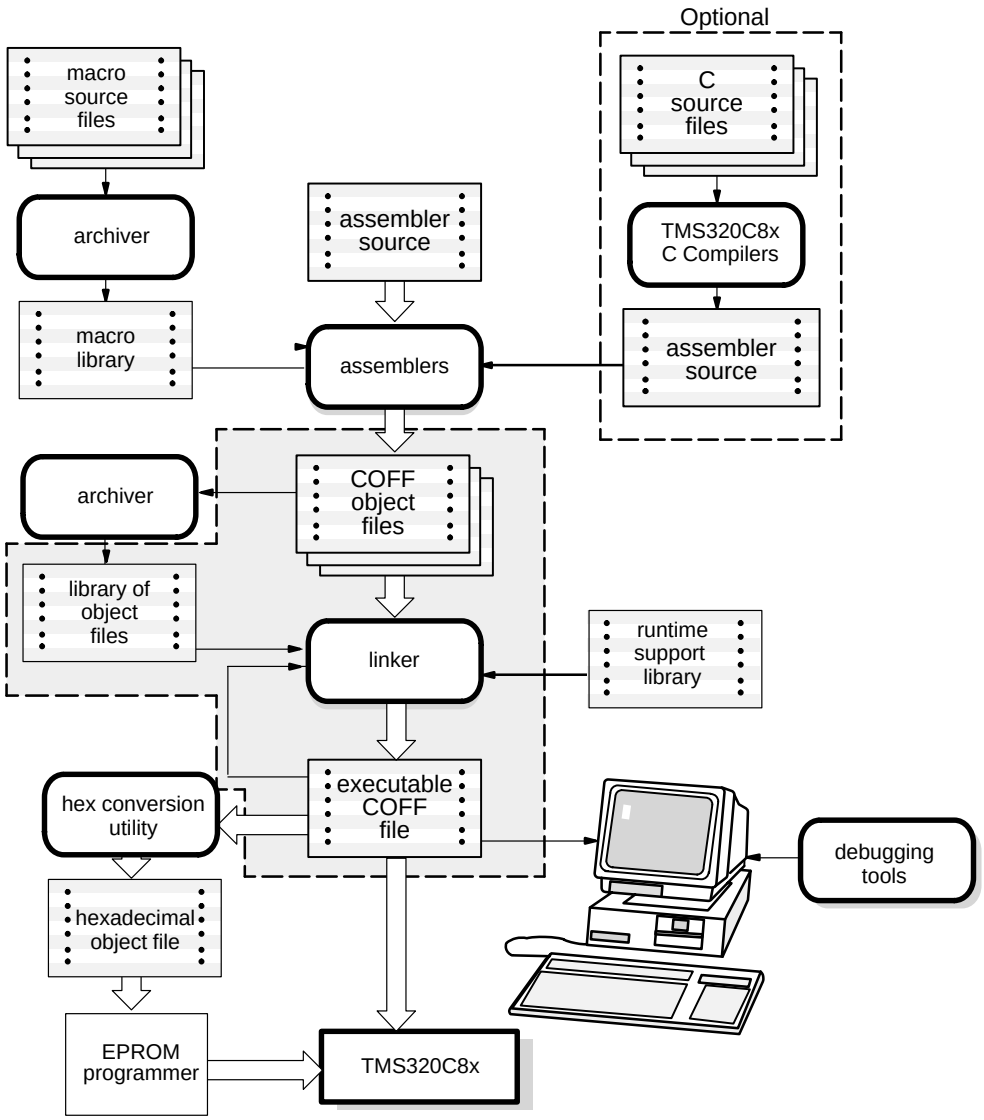
The linker creates executable modules by combining COFF object files. As the linker combines object files, it allocates sections into the target's configured memory, relocates symbols and sections, and resolves undefined external references.

The linker supports a C-like command language that controls memory configuration, output section definition, and address binding. The language supports expression assignment and evaluation and provides two powerful directives, MEMORY and SECTIONS, that allow you to:

- Define a memory model that conforms to target system memory
- Combine object file sections
- Allocate sections into specific areas of memory
- Define or redefine global symbols at link time

Figure 13–1 illustrates the linker's role in the assembly language development process. The linker accepts several types of files as input, including object files, command files, libraries, and partially linked files. The linker creates an executable COFF object module that can be downloaded to one of several development tools or executed by a TMS320C8x device.

Figure 13–1. Linker Development Flow



13.2 Invoking the Linker

The general syntax for invoking the linker is:

mvplnk [–*options*] *filename*$_1$ *. ... filename*$_n$

mvplnk is the command that invokes the linker.

options can appear anywhere on the command line or in a linker command file. (Options are discussed in Section 13.3, *Linker Options*.)

filenames can be object files, linker command files, or archive libraries. The default extension for all input files is *.obj*; any other extension must be explicitly specified. The linker can determine whether the input file is an object file or an ASCII file that contains linker commands. The default output filename is *a.out*.

There are three methods for invoking the linker:

- Specify options and filenames on the command line. This example links two files, file1.obj and file2.obj, and creates an output module named link.out.

```
mvplnk file1.obj file2.obj –o link.out
```

- Enter the **mvplnk** command with no filenames and no options; the linker will prompt for them:

```
Object files [.obj] :
Command files :
Output file [a.out] :
Options :
```

  - For *object files,* enter one or more object filenames. The default extension is *.obj.* Separate the filenames with spaces or commas; if the last character is a comma, the linker will prompt for an additional line of object filenames.
  - For *command files,* enter one or more command filenames.
  - The *output file* is the name of the linker output module. This overrides any –o options entered with any of the other prompts. If there are no –o options and you do not answer this prompt, the linker will create an object file with a default filename of *a.out*.
  - The *options* prompt is for additional options, although you can also enter them in a command file. Enter them with hyphens, just as you would on the command line.

- Put filenames and options in a linker command file. For example, assume the file linker.cmd contains the following lines:

```
–o link.out
file1.obj
file2.obj
```

  Now you can invoke the linker from the command line; specify the command filename as an input file:

```
mvplnk linker.cmd
```

  When you use a command file, you can also specify other options and files on the command line. For example, you could enter:

```
mvplnk –m link.map linker.cmd file3.obj
```

  The linker reads and processes a command file as soon as it encounters it on the command line, so it links the files in this order: file1.obj, file2.obj, and file3.obj. This example creates an output file called link.out and a map file called link.map.

13.3 Linker Options

Linker options control linking operations. They can be placed on the command line or in a command file. Linker options must be preceded by a dash (–). The order in which options are specified is unimportant, except for the –l and –i options. Options may be separated from arguments (if they have them) by an optional space. Table 13–1 summarizes the linker options.

Table 13–1. Linker Options Summary

Option	Description
–a	Produce an absolute, executable module. This is the default; if neither –a nor –r is specified, the linker acts as if –a were specified.
–ar	Produce a relocatable, executable object module.
–c	Link for TMS320C8x MP C code using the ROM autoinitialization model.
–cr	Link for TMS320C8x MP C code using the RAM autoinitialization model.
–e *global symbol*	Define a *global symbol* that specifies the primary entry point for the output module.
–f *fill value*	Set the default *fill value* for holes within output sections; *fill value* is a 32-bit constant.
–g *symbol*	Retain *symbol* as a global even if the –t or –h linker options are used.
–h	Make symbols defined with .global static.
–heap *size*	Set MP heap size (for the C memory pool command **malloc( )**) to *size* bytes and define a *global symbol* __SYSMEM_SIZE that specifies the MP's heap size. Default size is 1K bytes.
–i *dir*	Alter the library-search algorithm to look in *dir* before looking in the default location. This option must appear before the –l option.
–l *filename* †	Name an archive library file as linker input; *filename* is an archive library name.
–m *filename* †	Produce a map or listing of the input and output sections, including holes, and place the listing in *filename.*
–o *filename* †	Name the executable output module. The default filename is a*.out.*
–q	Request a quiet run (suppress the banner).
–pc	Link for TMS320C8x PP C code using the ROM autoinitialization model.
–pheap *size*	Set PP heap size (for the C memory pool command **malloc( )**) to *size* bytes and define a *global symbol* $_SYSMEM_SIZE that specifies heap size. Default size is 128 bytes.
–pstack *size*	Set PP C system stack size to *size* bytes and define a *global symbol* $_STACK_SIZE that specifies the PP's stack size. Default size is 128 bytes.
–r	Retain relocation entries in the output module.
–s	Strip symbol table information and line number entries from the output module.

Table 13–1. Linker Options Summary (Continued)

Option	Description
–stack *size*	Set MP C system stack size to *size* bytes and define a *global symbol* __STACK_SIZE that specifies the MP's stack size. Default size is 1K bytes.
–t *name*	Perform task level link. This results in a partial link that hides some global symbols.
–u *symbol*	Place an unresolved external *symbol* into the output module's symbol table.
–w	Generate warning when an output section that is not specified with the SECTIONS directive is created.
–x	Force rereading of libraries. Resolve back references.

† The *filename* must follow operating system conventions.

13.3.1 Relocation Capabilities (–a and –r Options)

One of the tasks the linker performs is *relocation.* Relocation is the process of adjusting all references to a symbol when the symbol's address changes. The linker supports two options (–a and –r) that allow you to produce an absolute or a relocatable output module.

❑ Producing an Absolute Output Module (–a Option)

When you use the –a option without the –r option, the linker produces an *absolute, executable* output module. Absolute files contain *no* relocation information. Executable files contain the following:

- Special symbols defined by the linker (subsection 13.12.4, *Symbols Defined by the Linker*, describes these symbols)
- An optional header that describes information such as the program entry point
- *No* unresolved references

This example links file1.obj and file2.obj and creates an absolute output module called a.out:

```
mvplnk -a file1.obj file2.obj
```

Note: –a and –r Options

If you do not use the –a or the –r option, the linker acts as if you specified –a.

Producing a Relocatable Output Module (–r Option)

When you use the **–r** option without the –a option, the linker retains relocation entries in the output module. If the output module will be relocated (at load time) or relinked (by another linker execution), use –r to retain the relocation entries.

The linker produces an *unexecutable* file when you use the –r option without –a. A file that is not executable does not contain special linker symbols or an optional header. The file may contain unresolved references, but these references do not prevent creation of an output module.

This example links file1.obj and file2.obj and creates a relocatable output module called a.out:

```
mvplnk –r file1.obj file2.obj
```

The output file a.out can be relinked with other object files or relocated at load time. (Linking a file that will be relinked with other files is called partial linking. For more information, see Section 13.14, *Partial (Incremental) Linking*.)

Producing an *Executable* Relocatable Output Module (–a –r)

If you invoke the linker with both the –a and –r options, the linker produces an *executable, relocatable* object module. The output file contains the special linker symbols, an optional header, and all resolved symbol references; however, the relocation information is retained.

This example links file1.obj and file2.obj and creates an executable, relocatable output module called xr.out:

```
mvplnk –a –r file1.obj file2.obj –o xr.out
```

Relocating or Relinking an Absolute Output Module

The linker issues a warning message (but continues executing) when it encounters a file that contains no relocation or symbol table information. Relinking an absolute file can be successful only if each input file contains no information that needs to be relocated (that is, each file has no unresolved references and is bound to the same virtual address that it was bound to when the linker created it).

13.3.2 C Language Options (–c, –pc, and –cr Options)

The –c and –cr options cause the linker to use linking conventions that are required by the TMS320C8x MP C compiler. The –pc option causes the linker to use linking conventions that are required by the TMS320C8x PP C compiler.

- The –c option tells the linker to use the ROM autoinitialization model for MP C code.
- The –cr option tells the linker to use the RAM autoinitialization model for MP C code.
- The –pc option tells the linker to use the ROM autoinitialization model for PP C code.

For more information about linking C code, see Section 1.7, *Linking C Code*. For more information about linking MP and PP C code, see Chapter 14, *Linking PP and MP Files: An Extended Example*.

13.3.3 Define an Entry Point (–e *global symbol* Option)

The memory address from which a program begins executing is called the **entry point.** When a loader loads a program into target memory, the program counter must be initialized to the entry point; the PC then points to the beginning of the program.

The linker can assign one of four possible values to the entry point. These values are listed below in the order in which the linker tries to use them. If you use one of the first three values, it must be an external symbol in the symbol table. Possible entry point values include:

1) The value specified by the –e option. The syntax is **–e** *global symbol*, where *global symbol* defines the entry point and must appear as an external symbol in one of the input files.
2) The value of symbol _c_int00 or $c_int00 (if present). The symbol _c_int00 (for the MP) or $c_int00 (for the PP) must be the entry point if you are linking code produced by the MP or PP C compilers.
3) The value of symbol _main (MP) or $main (PP) (if present)
4) Zero (default value)

This example links file1.obj and file2.obj. The symbol begin is the entry point; begin must be defined as external in file1 or file2.

```
mvplnk –e begin file1.obj file2.obj
```

13.3.4 Set Default Value (–f *cc* Option)

The **–f** option fills the holes formed within output sections or initializes uninitialized sections when they are combined with initialized sections. This allows you to initialize memory areas during link time without reassembling a source file. The argument *cc* is a 32-bit constant. If you do not use –f, the linker uses 0 as the default fill value, but will not automatically fill holes.

This example fills holes with the hexadecimal value ABCDABCD.

```
mvplnk -f 0ABCDABCDh file1.obj file2.obj
```

13.3.5 Make All .global Symbols Static (–h Option)

The **–h** option makes all symbols defined with the .global assembler directive static. This effectively "hides" the symbols, because static symbols are not visible to externally linked modules. This allows external symbols with the same name (in different files) to be treated as unique. The –h option will not modify global symbols defined with the .system assembler directive, or symbols that were made global with the –g linker option.

The –h option effectively nullifies all .global assembler directives. All symbols become local to the module in which they are defined, so no external references are possible. For example, assume file1.obj and file2.obj both define global symbols called EXT. By using the –h option and partial linking, you can link these files without conflict. The symbol EXT defined in file.out is treated separately from the symbol EXT defined in file2.out.

```
mvplnk –h –r file1.obj –o file1.out
mvplnk –h –r file2.obj –ofile2.out
mvplnk file1.out file2.out
```

13.3.6 Define MP Heap Size (–heap *size* Option)

The TMS320C8x MP C compiler uses an uninitialized section called .sysmem for the C runtime memory pool used by malloc( ). You can set the size in bytes of this memory pool at link time by using the **–heap** option. Specify the size as a constant immediately after the option:

```
mvplnk –heap 0x0800
      /* defines a 2k byte heap (.sysmem section)*/
```

The linker creates the .sysmem section only if there is a .sysmem section in an input file.

The linker also creates a global symbol __SYSMEM_SIZE and assigns it a value equal to the size of the heap. The default size is 1K bytes.

For more information about linking C code, see Section 1.7, *Linking C Code*.

13.3.7 Alter the Library Search Algorithm (–i *dir* Option/C_DIR)

Usually, when you want to specify a library as linker input, you simply enter the library name as you would any other input filename; the linker looks for the library in the current directory. For example, suppose the current directory contains the library object.lib. Assume that this library defines symbols that are referenced in the file file1.obj. This is how you link the files:

```
mvplnk file1.obj object.lib
```

If you want to use a library that is not in the current directory, use the –l (lowercase L) linker option. The syntax for this option is **–l** *filename.* The *filename* is the name of an archive library; the space between –l and the filename is optional.

You can augment the linker's directory search algorithm by using the **–i** linker option or the C_DIR environment variable. The linker searches for object libraries in the following order:

1) It searches directories named with the –i linker option.
2) It searches directories named with the environment variable C_DIR.
3) If C_DIR is not set, it searches directories named with the assembler's environment variable, A_DIR.
4) It searches the current directory.

The –i linker option

The –i linker option names an alternate directory that contains object libraries. The syntax for this option is **–i** *dir. dir* names a directory that contains object libraries; the space between –i and the directory name is optional. When the linker is searching for object libraries named with the –l option, it searches through directories named with –i first. Each –i option specifies only one directory, but you can use several –i options per invocation. When you use the –i option to name an alternate directory, it must precede the –l option on the command line or in a command file.

As an example, assume that there are two archive libraries called r.lib and lib2.lib, and that they are contained in directories /ld and /ld2 respectively. The command below shows how you would use the –i option to identify these paths.

```
mvplnk f1.obj f2.obj –i/ld –i/ld2 –lr.lib –llib2.lib
```

Environment variable (C_DIR)

An environment variable is a system symbol that you define and assign a string to. The linker uses an environment variable named **C_DIR** to name alternate directories that contain object libraries. The command for assigning the environment variable is:

```
setenv    C_DIR  "pathname;another pathname ... "
```

The *pathnames* are directories that contain object libraries. Use the –l option on the command line or in a command file to tell the linker which libraries to search for.

As an example, assume that two archive libraries called r.lib and lib2.lib reside in ld and ld2 directories. The code below shows the directories that r.lib and lib2.lib reside in, how to set the environment variable, and how to use both libraries during a link.

```
setenv C_DIR "/ldir;/ldir2"
mvplnk f1.obj f2.obj –l r.lib –l lib2.lib
```

Note that the environment variable remains set until you reboot the system or reset the variable by entering:

```
unsetenv    C_DIR
```

The assembler uses an environment variable named **A_DIR** to name alternate directories that contain copy/include files or macro libraries. If C_DIR is not set, the linker will search for object libraries in the directories named with A_DIR.

13.3.8 Create a Map File (–m *filename* Option)

The **–m** option creates a link map listing and puts it in *filename.* This map describes:

- ❑ Memory configuration
- ❑ Input and output section allocation
- ❑ The addresses of external symbols after they have been relocated

The map file contains the name of the output module and the entry point; it may also contain up to three tables:

- A table showing the new memory configuration if any non-default memory is specified.
- A table showing the linked addresses of each output section and the input sections that make up the output sections.
- A table showing each external symbol and its address. This table has two columns: the left column contains the symbols sorted by name, and the right column contains the symbols sorted by address.

This example links file1.obj and file2.obj and creates a map file called map.out:

```
mvplnk file1.obj file2.obj -m map.out
```

Section 14.5, *The Example Linker Map Files*, shows a sample map file.

13.3.9 Name an Output Module (–o *filename* Option)

The linker always creates an output module. If you do not specify a filename for the output module, the linker gives it the default name a.out. If you want to write the output module to a different file, use the **–o** option. The *filename* is the new output module name.

This example links file1.obj and file2.obj and creates an output module named run.out:

```
mvplnk -o run.out file1.obj file2.obj
```

13.3.10 Define PP Heap Size (–pheap *size* Option)

The TMS320C8x PP C compiler uses an uninitialized section called .psysmem for the C runtime memory pool used by malloc( ). You can set the size in bytes of this memory pool at link time by using the **–pheap** option. Specify the size as a constant immediately after the option:

```
mvplnk –pheap 0x0800
      /* defines a 2k heap (.psysmem section)*/
```

The linker creates the .psysmem section only if there is a .psysmem section in an input file.

The linker also creates a global symbol $_SYSMEM_SIZE and assigns it a value equal to the size of the heap. The default size for the PP is 128 bytes.

For more information about linking C code, see Section 1.7, *Linking C Code*.

13.3.11 Define PP Stack Size (–pstack *size* Option)

The TMS320C8x PP C compiler uses an uninitialized section, .pstack, to allocate space for the runtime stack. You can set the size in bytes of the .pstack section at link time with the **–pstack** option. Specify the *size* in bytes as a constant immediately after the option:

```
mvplnk –pstack 0x1000
      /* defines a 4K stack (.pstack section) */
```

If you specified a different stack size in an input section, the input section stack size is ignored. Any symbols defined in the input section remain valid; only the stack size will be different.

When the linker defines the .pstack section, it also defines a global symbol, $_STACK_SIZE, and assigns it a value equal to the size of the .pstack section. The default stack size for the PP is 128 bytes.

13.3.12 Specify a Quiet Run (–q Option)

The **–q** option suppresses the linker's banner when –q is the first option on the command line or in a command file. This option is useful for batch operation.

13.3.13 Strip Symbolic Information (–s Option)

The **–s** option creates a smaller output module by omitting symbol table information and line number entries. The –s option is useful for production applications when you must create the smallest possible output module.

This example links file1.obj and file2.obj and creates an output module, stripped of line numbers and symbol table information, named nosym.out:

```
mvplnk –o nosym.out –s file1.obj file2.obj
```

Note that using the –s option limits later use of a symbolic debugger and may prevent a file from being relinked.

13.3.14 Define MP Stack Size (–stack *size* Option)

The TMS320C8x MP C compiler uses an uninitialized section, .stack, to allocate space for the runtime stack. You can set the size in bytes of the .stack section at link time with the **–stack** option. Specify the *size* as a constant immediately after the option:

```
mvplnk –stack 0x1000
      /* defines a 4K stack (.stack section) */
```

If you specified a different stack size in an input section, the input section stack size is ignored. Any symbols defined in the input section remain valid; only the stack size will be different.

When the linker defines the .stack section, it also defines a global symbol, __STACK_SIZE, and assigns it a value equal to the size of the .stack section. The default MP stack size is 1K bytes.

13.3.15 Task Level Linking (–t Option)

The C language provides for three basic symbol scope (or visibility) rules. First, auto variables are those symbols defined within a function and they are visible only within that function. Second, static variables are defined with the static keyword, and are visible only within the file they are defined in. Third, global symbols are defined outside of a function and without a static keyword. Global symbols can be visible to any file.

In some instances it may be useful to have another level of visibility that is similar to global symbols, but is restricted to a subset of files. The –t linker option is used to achieve this extra level of visibility, called task level visibility.

The mechanism used to achieve task level linking is as follows. The assembler has two directives which it uses to indicate that a symbol is global, .global and .system. These directives are identical in behavior, unless the –t (or –h) linker option is used when linking the files that define the symbols.

The –t or (–h) option will cause the .global defined symbols to become static. The .system defined symbols will remain global.

By using the .system and .global assembler directives and the –t linker option, task level visibility can be achieved for a set of files. The set of files would be those linked in the task level link. The syntax of the –t option is **–t** *taskname*. The taskname must be a unique symbol assigned to the task. The symbol must be eight characters or less.

The **–t** *taskname* linker option will cause the following things to occur:

1) The –r linker option will be invoked to retain relocation entries for further links.

2) The –h linker option will be invoked to change defined .global symbols into static symbols.

3) A new filename symbol will be created with the name *taskname*. All the data symbols converted to static from global will be associated with this new filename symbol. This will allow you to reference the data symbols in the debugger (assuming you compiled with debug on) using the filename.variable syntax. For example: if you invoked the –t option as –t task1, and a variable var1 was converted from global to static, then you could refer to var1 as task1.var1 in the debugger.

   Function names that become static will remain associated with the file they were defined in. The filename.var syntax can still be used to reference function names, for example, a function f defined in T1.c can be referenced as T1.f.

Note: Task Names Must Be Unique

The taskname used with the –t *taskname* linker option should not have the same name as any symbol in the output file .If the taskname symbol is not unique, then the filename.var syntax will not work correctly.

4) Any undefined task level symbols (i.e. those being converted from global to static) will remain global.

5) If an entry point symbol is defined, then –t will create two new symbols: $ep_*taskname* and _ep_*taskname* and will equate these symbols to the entry point symbol. This allows two separate tasks to have main or c_int00 routines without conflicting, and allows other routines to distinguish between the two with the $ep or _ep symbols.

6) All linker defined symbols that normally occur with the –a linker option or by default when –r or –t is not specified will be defined. The symbols include the stack size, heap size, and cinit symbols.

Example

This example links four files file1.obj, file2.obj, file3.obj, and file4.obj and uses task level links to provide task visibility for the symbol taskvar.

file1.asm

```
taskvar .equ 1
sysvar  .equ 2
   .global taskvar
   .system sysvar
   .word   taskvar
   .word   sysvar
```

file2.asm

```
   .global taskvar
   .system sysvar
   .word   taskvar
   .word   sysvar
```

file3.asm

```
taskvar .equ 3
   .global taskvar
   .system sysvar
   .word   taskvar
   .word   sysvar
```

file4.asm

```
   .global taskvar
   .system sysvar
   .word   taskvar
   .word   sysvar
```

The following command will task level link file1.obj and file2.obj. The output file task1.out will contain a static taskvar and a .system defined global sysvar.

```
mvplnk –t task1 file1.obj file2.obj –o task1.out
```

The following command will task level link file3.obj and file4.obj. The output file task2.out will contain a static taskvar (different from the taskvar in task1.out) and a .system global sysvar, which is unresolved.

```
mvplnk –t task2 file3.obj file4.obj –o task2.out
```

The following command will combine the output files from the two task level links. The separate taskvar definitions will not conflict, and the sysvar unresolved reference in task2.out will be resolved by the global definition of sysvar in task1.out.

```
mvplnk task1.out task2.out –o example.out
```

13.3.16 Introduce an Unresolved Symbol (–u *symbol* Option)

The **–u** option introduces an unresolved symbol into the linker's symbol table. This forces the linker to search through a library and include the member that defines the symbol. Note that the linker must encounter the –u option *before* it links in the member that defines the symbol.

For example, suppose a library named cio.lib contains a member that defines the symbol exit; none of the object files you are linking reference exit. However, suppose you plan to relink the output module, and you would like to include the library member that defines exit in this link. Using the –u option as shown below forces the linker to search cio.lib for the member that defines exit and to link in the member.

```
mvplnk -u exit file1.obj file2.obj cio.lib
```

If you do not use –u, this member is not included, because there is no explicit reference to it in file1.obj or file2.obj.

13.3.17 Warning Switch (–w Option)

The –w option generates a warning message if an output section that is not explicitly specified with the SECTIONS directive is created. For example:

```
–[ fl.asm ]–

.sect ”xsect”
.word 0

–[ link.cmd ]–

SECTIONS
{
   <no output section specifications reference xsect>
}
```

The linker would create an output section called xsect that consists of the input section xsect from other object files, and use the default allocation rules to allocate this output section into memory. If you used the –w switch when linking, this example would generate the message:

```
>> warning: creating output section xsect without SECTIONS
specification
```

The –w option will also cause the linker to give warning messages about any undefined task-level global variables in a task level link.

13.3.18 Exhaustively Read Libraries (–x Option)

The linker normally reads input files, including archive libraries, only once when they are encountered on the command line or in the command file. When an archive is read, any members that resolve references to undefined symbols are included in the link. If an input file later references a symbol defined in a previously read archive library (this is called a *back reference*), the reference will not be resolved.

With the **–x** option, you can force the linker to repeatedly reread all libraries. The linker will continue to reread libraries until no more references can be resolved. For example, if a.lib contains a reference to a symbol defined in b.lib, and b.lib contains a reference to a symbol defined in a.lib, you can resolve the mutual dependencies by listing one of the libraries twice, as in:

```
mvplnk –la.lib –lb.lib –la.lib
```

or you can force the linker to do it for you:

```
mvplnk –x –la.lib –lb.lib
```

Linking using the –x option may be slower, so you should use the option only as needed.

13.4 Linker Command Files

Linker command files allow you to put linking information in a file; this is useful when you often invoke the linker with the same information. Linker command files are also useful because they allow you to use the MEMORY and SECTIONS directives to customize your application. You must use these directives in a command file; you cannot use them on the command line. Command files are ASCII files that contain one or more of the following:

- Input filenames, which specify object files, archive libraries, or other command files. (If a command file calls another command file as input, this statement must be the *last* statement in the calling command file. The linker does not return from called command files.)
- Linker options, which can be used in the command file in the same manner in which they are used on the command line.
- The MEMORY directive, which defines the target memory configuration, and the SECTIONS directive, which controls how sections are built and allocated.
- Assignment statements, which define and assign values to global symbols.

To invoke the linker with a command file, enter the **mvplnk** command and follow it with the name of the command file:

mvplnk *command filename*

The linker processes input files in the order that it encounters them. If the linker recognizes a file as an object file, it links it. Otherwise, it assumes that a file is a command file and begins reading and processing commands from it. Note that command filenames are upper/lower-case sensitive, regardless of the system used.

Example 13–1 shows a sample linker command file called link.cmd. (Subsection 12.3.2, *Placing Sections in the Memory Map*, contains another example of a linker command file.)

Example 13–1. Linker Command File

```
/************************************************/
/*            Sample Linker Command File        */
/************************************************/
a.obj              /*  First input filename          */
b.obj              /*  Second input filename         */
–o prog.out        /*  Option to specify output file */
–m prog.map        /*  Option to specify map file    */
```

The sample file in Example 13–1 contains only filenames and options. (Note that you can place comments in a command file by delimiting them with /* and */.) To invoke the linker with this command file, enter:

```
mvplnk link.cmd
```

You can place other parameters on the command line when you use a command file, for example:

```
mvplnk –r link.cmd c.obj d.obj
```

The linker processes the command file as soon as it encounters it, so a.obj and b.obj are linked into the output module before c.obj and d.obj.

You can specify multiple command files. If, for example, you have a file called names.lst that contains filenames and another file called dir.cmd that contains linker directives, you could enter:

```
mvplnk names.lst dir.cmd
```

One command file can call another command file; command file nesting is limited to 16 levels. If a command file calls another command file as input, the statement must be the *last* statement in the calling command file.

Blanks and blank lines that appear in a command file are insignificant except as delimiters. This also applies to the format of linker directives in a command file. Example 13–2 shows a sample command file that contains linker directives. (Linker directive formats are discussed in later sections.)

Example 13–2. Command File With Linker Directives

```
/*****************************************************/
/*    Sample Linker Command File with Directives     */
/*****************************************************/
a.obj b.obj c.obj           /* Input filenames       */
–o prog.out –m prog.map     /* Options               */

MEMORY                      /* MEMORY directive      */
{
  RAM:  origin = 0x02000000 length = 0x8000
  ROM:  origin = 0x03000000 length = 0x4000
}

SECTIONS                    /* SECTIONS directive    */
{
  .text:  > ROM
  .data:  > ROM
  .bss:   > RAM
}
```

The following names are reserved as keywords for linker directives. Do not use them as symbol or section names in a command file.

align	l (lowercase L)	page
ALIGN	len	PAGE
attr	length	pass
ATTR	LENGTH	PASS
block	load	range
BLOCK	LOAD	run
COPY	MEMORY	RUN
DSECT	NOLOAD	SECTIONS
f	o	spare
fill	org	type
group	origin	TYPE
GROUP	ORIGIN	UNION

Constants in command files

Constants can be specified with either of two syntax schemes: the scheme used for specifying decimal, octal, or hexadecimal constants used in the assembler (see Section 8.2, *Constants*) or the scheme used for integer constants in ’C’ syntax.

Examples:

	Decimal	**Octal**	**Hexadecimal**
Assembler Format:	32	40q	20h
’C’ Format:	32	040	0x20

13.5 Object Libraries

An object library is a partitioned archive file that contains complete object files as members. Usually, a group of related modules are grouped together into a library. When you specify an object library as linker input, the linker includes any members of the library that define existing unresolved symbol references. You can use the TMS320C8x archiver to build and maintain libraries; Chapter 16, *Archiver Description* contains more information about the archiver.

Using object libraries can reduce link time and the size of the executable module. If a normal object file that contains a function is specified at link time, it is linked whether it is used or not; however, if that same function is placed in an archive library and the archive is included at link time, it is included only if it is referenced.

The order in which libraries are specified is important, because the linker includes only those members that resolve symbols that are undefined when the library is searched. The same library can be specified as often as necessary; it is searched each time it is included. Optionally, the –x option can be used to force the linker to reread libraries until it can resolve no more references. A library has a table that lists all external symbols defined in the library; the linker searches through the table until it determines that it cannot use the library to resolve any more references.

This example links several files and libraries. Assume the following:

- Input files f1.obj and f2.obj both reference an external function named clrscr.
- Input file f1.obj references the symbol origin.
- Input file f2.obj references the symbol fillclr.
- Library libc.lib, member 0, contains a definition of origin.
- Library liba.lib, member 3, contains a definition of fillclr.
- Member 1 of both libraries defines clrscr.

If you enter mvplnk f1.obj liba.lib f2.obj libc.lib, then:

- Member 1 of liba.lib satisfies both references to clrscr because the library is searched and clrscr is defined before f2.obj references it.
- Member 0 of libc.lib satisfies the reference to origin.
- Member 3 of liba.lib satisfies the reference to fillclr.

If, however, you enter mvplnk f1.obj f2.obj libc.lib liba.lib, the references to clrscr are satisfied by member 1 of libc.lib.

If none of the linked files reference symbols defined in a library, you can use the –u option to force the linker to include a library member. The next example creates an undefined symbol rout1 in the linker's global symbol table:

```
mvplnk –u rout1 libc.lib
```

If any members of libc.lib define rout1, the linker includes those members. Note that it is not possible to control the allocation of individual library members; sections from individual library members are controlled by generic entries in the SECTIONS directive.

Subsection 13.3.7, *Alter the Library Search Algorithm (–i dir Option/C_DIR)*, describes methods for specifying directories that contain object libraries.

13.6 The MEMORY Directive

The linker determines where output sections should be allocated into memory; the linker must have a model of target memory to accomplish this task. The MEMORY directive allows you to specify a model of target memory so that you can define the types of memory your system contains and the address ranges they occupy. The linker maintains the model as it allocates output sections and uses the model to determine which memory locations can be used for object code.

The memory configurations of ’C8x systems differ from application to application. The MEMORY directive allows you to specify a variety of configurations. After you use the MEMORY directive to define a memory model, you can use the SECTIONS directive to allocate output sections into defined memory.

13.6.1 Default Memory Model

The entire MVP memory map is a single 32-bit memory space common to all processors. The MVP can address 2^{32} bytes directly. Addresses less than 0x0200 0000 are assigned to internal memory and can be addressed as such, and addresses from 0x0200 0000 to 0xFFFF FFFF are assigned to external memory and must be addressed as such.

Each processor has five associated RAMs. The MP has two instruction cache RAMs, two data cache RAMs, and one parameter RAM. Each PP has one instruction cache RAM, one parameter RAM, and three data RAMs. If the system is a one-PP MVP, there are two extra uncached data banks (PP0 data RAM 3 and 4) for PP0.

Data RAMs are the main areas for the PPs to store the data they are processing. The data RAMs are situated in the lower address space.

Parameter RAMS provide transfer tables for interrupt routines, and they include the system stack, packet transfer address pointers, task parameter passing, and interprocessor command messages.

Instruction caches contain cached executable code. For more information about MVP memory organization, see Chapter 3, *Understanding the MVP Memory Organization* in the *MVP System-Level Synopsis.*

Most program sections will be allocated in external memory (addresses 0x200 0000 – FFFF FFFF), except for a few uninitialized sections that are used by the PPs.

If you do not use the MEMORY directive, the linker uses a default memory model specific to the TMS320C8x device. For more information about the default memory model, see subsection 13.10.1, *Allocation Algorithm*.

13.6.2 MEMORY Directive Syntax

The MEMORY directive identifies ranges of memory that are physically present in the target system and can be used by a program. Each memory range has a *name,* a *starting address,* and a *length.*

When you use the MEMORY directive, be sure to identify *all* the memory ranges that are available to load object code into. Memory that is defined by the MEMORY directive is *configured memory*; any memory that you do not explicitly account for with the MEMORY directive is *unconfigured memory***.** The linker does not place any part of a program into unconfigured memory. You can represent nonexistent memory spaces by simply not including an address range in a MEMORY directive statement.

The MEMORY directive is specified in a command file by the word MEMORY (uppercase), followed by a list of memory range specifications enclosed in braces. The MEMORY directive in Example 13–3 defines a system that has 4K bytes of ROM at address 0x00000000 in memory, 2K bytes of RAM at address 0x01000000, and 512 bytes at address 0x02000000 in external memory.

Example 13–3. The MEMORY Directive

```
/*--------------------------------------------------------------------------*/
/*  Sample command file with MEMORY directive                               */
/*--------------------------------------------------------------------------*/
file1.obj file2.obj                                    /*  Input files   */
-o prog.out                                            /*  Options       */

MEMORY
{
    DRAM   :  origin = 0x00000000,  length = 0x1000
    PRAM   :  origin = 0x01000000,  length = 0x0800
    EXTMEM :  origin = 0x02000000,  length = 0x80000
}
```

MEMORY directive

names **origins** **lengths**

Now you can use the SECTIONS directive to tell the linker where to link the sections. For example, you could allocate the .text and .ptext sections into the area named EXTMEM and allocate the .pbss section into DRAM or PRAM.

The general syntax for the MEMORY directive is:

MEMORY
{
name 1 [(attr)] **: origin =** *constant,* **length =** *constant;*
name n [(attr)] **: origin =** *constant,* **length =** *constant;*
}

name names a memory range. A memory name may be one to eight characters; valid characters include A–Z, a–z, $, ., and _. The names have no significance to the program; they simply identify memory ranges for the linker. Memory range names are internal to the linker and are not retained in the output file or in the symbol table. All memory ranges must have unique names and must not overlap.

attr specifies 1 to 4 optional attributes that are associated with the named range. Valid attributes include **R** (readable memory), **W** (writable memory), **X** (executable memory), and **I** (initializable memory); attributes must be enclosed in parentheses. If you do not specify any attributes for a memory range, the range *has all four attributes.* All memory in the default model has all four attributes. The following example specifies a memory range with the R and X attributes:

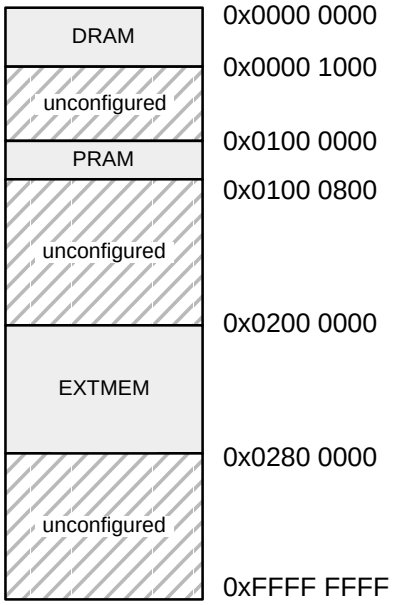
```
MEMORY
{   ROM (RX) : o = 0x03000000, l = 01000h   }
```

origin specifies the starting address of a memory range. It may be entered as *origin, org,* or *o.* The value, specified in bytes, is a 32-bit constant and may be decimal, octal, or hexadecimal.

length specifies the length of a memory range. It may be entered as *length, len,* or *l.* The value, specified in bytes, is a 32-bit constant and may be decimal, octal, or hexadecimal.

Figure 13–2 illustrates the memory map defined by Example 13–3.

Figure 13–2. Memory Map Defined in Example 13–3



13.7 The SECTIONS Directive

The SECTIONS directive tells the linker how to combine sections from input files into sections in the output module and where to place the output sections in memory. In summary, the SECTIONS directive:

- Describes how input sections are combined into output sections
- Defines output sections in the executable program
- Specifies where output sections are placed in memory (in relation to each other and to the entire memory space)
- Permits renaming of output sections

13.7.1 Default Sections Configuration

If you do not specify a SECTIONS directive, the linker uses a default algorithm for combining and allocating the sections. Section 13.10, *Default Allocation Algorithm*, describes this algorithm in detail.

13.7.2 SECTIONS Directive Syntax

Note: Compatibility With Previous Versions

In previous versions of the linker, many of these constructs were specified differently. The linker accepts any of the older forms.

The SECTIONS directive is specified in a command file by the word SECTIONS (uppercase), followed by a list of output section specifications enclosed in braces.

The general syntax for the SECTIONS directive is:

```
SECTIONS
 {
        name1 : [ property, property, property, ... ]
        name2 : [ property, property, property, ... ]
        name3 : [ property, property, property, ... ]
        . . .
}
```

Each section specification, beginning with *name*, defines an output section. (An output section is a section in the output file.) After the section *name* is a list of properties that define the sections contents and how it is allocated. The properties may be separated by optional commas. Possible properties for a section are:

- **Load allocation**, which defines where in memory the section is to be loaded
  Syntax: **load =** *allocation* or
  allocation or
  > *allocation*

- **Run allocation**, which defines where in memory the section is to be run
  Syntax: **run =** *allocation* or
  run > *allocation*

- **Input sections**, which defines the input sections comprising the section
  Syntax: **{** input_sections **}**

- **Section type**, which defines flags for special section types
  Syntax: **type = COPY** or
  type = DSECT or
  type = NOLOAD or
  type = PASS

  For more information on section types, see Section 13.11, *Special Section Types (DSECT, COPY, NOLOAD, and PASS)*.

- **Fill value**, which defines the value used to fill uninitialized holes.
  Syntax: **fill =** *value* or
  name: ... { ... } **=** *value*
  For more information on creating and filling holes, see Section 13.13, *Creating and Filling Holes*.

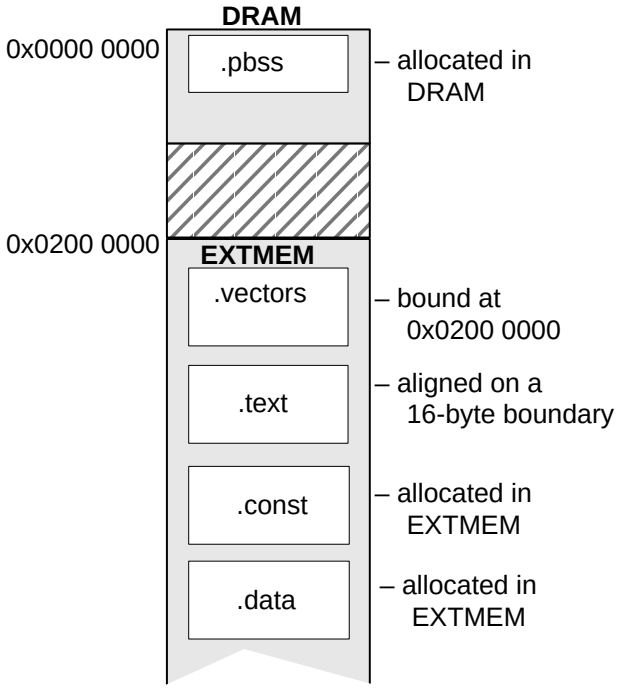
Example 13–4 shows a SECTIONS directive in a sample linker command file. Figure 13–3 shows how these sections are allocated in memory.

Example 13–4. The SECTIONS Directive

```
/*--------------------------------------------------------------------------*/
/*  Sample command file with SECTIONS directive                             */
/*--------------------------------------------------------------------------*/
file1.obj file2.obj                                    /*  Input files   */
-o prog.out                                            /*  Options       */

SECTIONS                  <-- SECTIONS directive
{
      .text    : load = EXTMEM, align = 16
      .const   : load = EXTMEM, run = DRAM
      .pbss    : load = DRAM
      .vectors : load = 0x02000000
      {
         t1.obj(vecs1)
         t2.obj(vecs2)
      }
      .data    : load = EXTMEM
}
```


Figure 13–3 shows the five output sections defined by the SECTIONS directive in Example 13–4; .vectors, .text, .const, .pbss, and .data.

Figure 13–3. Section Allocation Defined by Example 13–4



The **.pbss** section combines the .pbss sections from file1.obj and file2.obj.

The **.vectors** section is composed of the vecs1 section from t1.obj and the vecs2 section from t2.obj.

The **.text** section combines the .text sections from file1.obj and file2.obj. The linker combines all sections named .text into this section

The **.const** section is combines the .const sections from file1.obj and file2.obj. The application must relocate the section to DRAM to run.

The **.data** section combines the .data sections from file1.obj and file2.obj.

13.7.3 Specifying the Address of Output Sections (Allocation)

The linker assigns each output section two locations in target memory: the location where the section will be loaded and the location where it will be run. Usually, these are the same, and you can think of each section as having only a single address. In any case, the process of locating the output section in the target's memory and assigning its address(es) is called allocation. For more information about using separate load and run allocation, see Section 13.8, *Specifying a Section's Runtime Address*.

If you do not tell the linker how a section is to be allocated, it uses a default algorithm to allocate the section. You can override this default allocation for a section by defining it within a SECTIONS directive and providing instructions on how to allocate it.

You control allocation by specifying one or more allocation parameters. Each parameter consists of a keyword, an optional equal sign or greater-than sign, and a value optionally enclosed in parentheses. If load and run allocation is separate, all parameters following the keyword LOAD apply to load allocation, and those following RUN apply to run allocation. Possible allocation parameters are:

Binding allocates a section at a specific address.

```
.text: load = 0x1000
```

Memory allocates the section into a range defined in the MEMORY directive with the specified name or attributes.

```
.text: load > ROM
```

Alignment specifies that the section should start on an address boundary.

```
.text: align = 0x80
```

Blocking specifies that the section must fit between two address boundaries: if the section is too big, it will start on an address boundary.

```
.text: block(0x80)
```

For the load (usually the only) allocation, you may simply use a greater-than sign and omit the LOAD keyword:

```
.text: > EXTMEM            .text: {...} > EXTMEM
.text: > 0x200000
```

If more than one parameter is used, you can string them together as follows:

```
.text: > EXTMEM align 16
```

Or, if you prefer, use parentheses for readability:

```
.text: load = (EXTMEM align(16))
```

Binding

You can supply a specific starting address for an output section by following the section name with an address:

```
.text: 0x02000000
```

This example specifies that the .text section must begin at location 2000000h. The binding address must be a 32-bit constant.

Output sections can be bound anywhere in configured memory (assuming there is enough space), but they cannot overlap. If there is not enough space to bind a section to a specified address, the linker issues an error message.

Note: Binding and Alignment of Named Memory Are Incompatible

You cannot bind a section to an address if you use alignment or named memory. If you try to do this, the linker issues an error message.

Named memory

You can allocate a section into a memory range that is defined by the MEMORY directive. This example names ranges and links sections into them:

```
MEMORY
{
   DRAM (RW) :  origin = 0x00000000    length = 0x1000
   ROM (RIX):   origin = 0x03000000,   length = 0x3000
}
SECTIONS
{
   .text  :       > ROM
   .data :        > ROM ALIGN(128
   .pbss  :       > DRAM
}
```

In this example, the linker places .text and .data into the area called ROM. The .pbss output sections are allocated into DRAM. You can align a section within a named memory range; the .data section is aligned on a 128-byte boundary within the ROM range.

Similarly, you can link a section into an area of memory that has particular attributes. To do this, specify a set of attributes (enclosed in parentheses) instead of a memory name. Using the same MEMORY directive declaration, you can specify:

```
SECTIONS
{
   .text:     > (X)    /* .text  --> executable memory      */
   .data:     > (RI)   /* .data  --> read or init memory    */
   .pbss :    > (RW)   /* .pbss  --> read or write memory   */
}
```

In this example, the .text output section is linked into the ROM area, because it has the X attribute. The .data section is allocated into ROM, because it has the R and I attributes. The .bss output section, must go into the DRAM area, because only DRAM is declared with the W attribute.

You cannot control where in a named memory range a section is allocated, although the linker uses lower memory addresses first and avoids fragmentation when possible. In the preceding examples, assuming no other sections had been bound to addresses that would interfere with this allocation process, the .text section would start at address 0x03000000. If a section must start on a specific address, use binding instead of named memory.

Alignment and blocking

You can tell the linker to place an output section at an address that falls on an n-byte boundary, where n is a power of 2. For example:

```
.text: load = align(128)
```

allocates .text so that it falls on a page boundary.

Blocking is a weaker form of alignment that places a section so that it is allocated anywhere **within** a block of size *n* bytes. If the section is larger than the block size, the section will begin on that boundary. As with alignment, *n* must be a power of 2. For example:

```
bss: load = block(0x40)
```

allocates .bss so that the section either is contained in a single 64-byte page or begins on a page.

You can use alignment or blocking alone or in conjunction with a memory area, but alignment and blocking cannot be used together.

13.7.4 Specifying Input Sections

An input section specification identifies the sections from input files that are combined to form an output section. The linker combines input sections by concatenating them in the order in which they are specified. The size of an output section is the sum of the sizes of the input sections that make up the output section.

Example 13–5 shows the most common type of section specification; note that *no* input sections are listed.

Example 13–5. The Most Common Method of Specifying Section Contents

```
SECTIONS
{
   .text:
   .data:
   .bss:
}
```

In Example 13–5, the linker takes all the .text sections from the input files and combines them into the .text output section. The linker concatenates the .text input sections in the order that it encounters them in the input files. The linker performs similar operations with the .data and .bss sections. You can use this type of specification for *any* output section.

You can explicitly specify the input sections that form an output section. Each input section is identified by its filename and section name:

```
SECTIONS
{
   .text :            /* Build .text output section     */
   {
      f1.obj(.text)   /* Link .text section from f1.obj  */
      f2.obj(sec1)    /* Link sec1 section from f2.obj   */
      f3.obj          /* Link ALL sections from f3.obj   */
      f4.obj(.text,sec2)
                      /* Link .text and sec2 from f4.obj */
   }
}
```

Note that it is not necessary for input sections to have the same name as each other or as the output section they become part of. If a file is listed with no sections, *all* of its sections are included in the output section. If any additional input sections have the same name as an output section but are not explicitly specified by the SECTIONS directive, they are automatically linked in at the end of the output section. For example, if the linker found more .text sections in the preceding example, and these .text sections *were not* specified anywhere in the SECTIONS directive, the linker would concatenate these extra sections after f4.obj(sec2).

The specifications in Example 13–5 are actually a shorthand method for the following:

```
SECTIONS
{
   .text:{ *(.text) }
   .data:{ *(.data) }
   .bss: { *(.bss)  }
}
```

The *(.text) means *the unallocated .text sections from all the input files.* This format is useful when:

- You want the output section to contain all input sections that have a certain name, but the output section name is different from the input sections' name.
- You want the linker to allocate the input sections *before* it processes additional input sections or commands within the braces.

Here's an example that uses this method:

```
SECTIONS
{
   .text  :  {
                    abc.obj(xqt)
                       *(.text)
             }
   .data  :  {
                    *(.data)
                    fil.obj(table)
             }
}
```

In this example, the .text output section contains a named section xqt from file *abc.obj*, which is followed by *all* the .text input sections. The .data section contains *all* the .data input sections, followed by a named section *table* from the file *fil.obj*. Note that this method includes all the *unallocated* sections. For example, if one of the .text input sections was already included in another output section when the linker encountered *(.text), the linker could not include that first .text input section in the second output section.

13.8 Specifying a Section's Runtime Address

At times you may want to load code into one area of memory and run it in another. For example, you may have performance-critical code in a ROM-based system. The code must be loaded into ROM, but would run faster if it were in RAM.

The linker provides a simple way to specify this. In the SECTIONS directive, you can direct the linker to allocate a section twice: once to set its load address and again to set its run address. For example:

```
.fir: load = ROM, run = RAM
```

Use the *load* keyword for the load address and the *run* keyword for the run address.

13.8.1 Specifying Two Addresses

The load address determines where a loader will place the raw data for the section. Any references to the section (such as labels in it) refer to its run address. The application must copy the section from its load address to its run address; this does NOT happen automatically when you specify a separate run address.

If you provide only one allocation (either load or run) for a section, the section is allocated only once and will load and run at the same address. If you provide both allocations, the section is actually allocated as if it were two different sections of the same size. This means that both allocations occupy space in the memory map and cannot overlay each other or other sections. (The UNION directive provides a way to overlay sections; see subsection 13.9.1, *Overlaying Sections With the UNION Directive*.)

If either the load or run address has additional parameters, such as alignment or blocking, list them after the appropriate keyword. Everything related to allocation after the keyword *load* affects the load address until the keyword *run* is encountered, then everything affects the run address. The load and run allocations are completely independent, so any qualification of one (such as alignment) has no effect on the other. You may also specify run first, then load. Use parentheses to improve readability. Following are examples:

```
.data: load = EXTMEM, align = 32, run = DRAM
```

(align applies only to load)

```
.data: load = (EXTMEM align 32), run = DRAM
```

(identical to previous example)

```
.data: run   =  DRAM, align 32,
       load  =  align 16
```

(align 32 in DRAM for run; align 16 anywhere for load)

13.8.2 Uninitialized Sections

Uninitialized sections (such as .bss) are not loaded, so the only address of significance is the run address. The linker allocates uninitialized sections only once: if you specify both run and load addresses, the linker warns you and ignores the load address. Otherwise, if you specify only one address, the linker treats it as a run address, regardless of whether you call it load or run. For example:

```
.bss: load = 0x1000, run = EXTMEM
```

A warning is issued, load is ignored, and space is allocated in RAM. All of the following examples have the same effect. The .bss section is allocated in EXTMEM.

```
.bss: load = EXTMEM
.bss: run = EXTMEM
.bss: > EXTMEM
```

13.8.3 Referring to the Load Address by Using the .label Directive

Any reference to a normal symbol in a section refers to its runtime address. However, it may be necessary at runtime to refer to a load-time address. In particular, the code that copies a section from its load address to its run address must know where it was loaded. The .label directive in the assembler defines a special symbol that refers to the load address of the section. Thus, whereas normal symbols are relocated with respect to the run address, .label symbols are relocated with respect to the load address. For more information on the .label directive, see page CG:9-48.

Example 13–6. Copying a Section From ROM to RAM

```
;-----------------------------------------------------------------
;  define a section to be copied from external to internal memory
;-----------------------------------------------------------------
          .sect  ".fir"
          .label fir_src          ; load address of section
fir:                              ; run address of section
          <code here>             ; code for the section

       .label fir_end             ; load address of section end
```

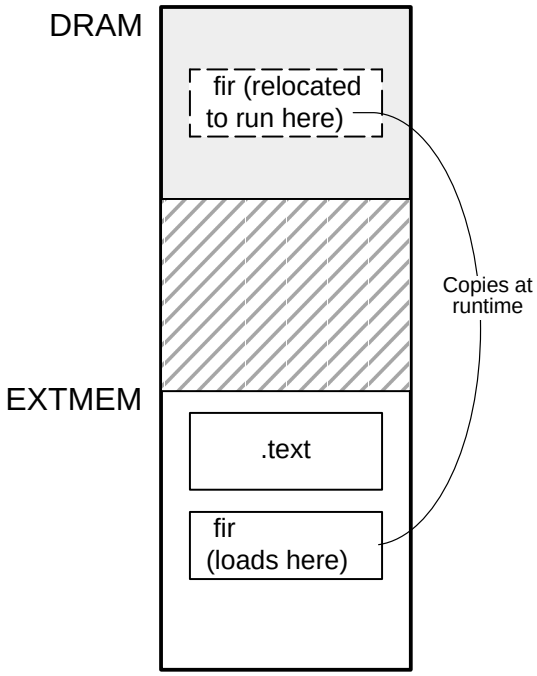
```
/*****************************************************************/
/*  copy .fir section into on-chip RAM with C code               */
/*****************************************************************/
memcpy( fir, fir_src, fir_end - fir_src );

/*****************************************************************/
/*  jump to section, now in on-chip RAM with C code              */
/*****************************************************************/
       fir();
```

Linker Command File

```
/*****************************************************************/
/* PARTIAL LINKER COMMAND FILE FOR FIR EXAMPLE                   */
/*****************************************************************/
MEMORY
{
   DRAM   :  origin = 0x00000000, length = 0x1000
   EXTMEM :  origin = 0x02000000, length = 0x800000
}

SECTIONS
{
   .text: load = EXTMEM
   .fir:  load = EXTMEM, run = DRAM
}
```

Figure 13–4 illustrates the runtime execution of this example.

Figure 13–4. Runtime Execution of Example 13–6



13.9 Using UNION and GROUP Statements

Two SECTIONS statements allow you to conserve memory: GROUP and UNION. Unioning sections causes the linker to allocate the same run address to the sections. Grouping sections causes the linker to allocate them contiguously in memory.

13.9.1 Overlaying Sections With the UNION Directive

For some applications, you may want to allocate more than one section to run at the same address; for example, you may have several routines you want in on-chip RAM at various stages of the program's execution. Or you may want several data objects that you know will not be active at the same time to share a block of memory. The UNION statement within the SECTIONS directive provides a way to allocate several sections at the same run address.

Example 13–7. The Form of the UNION Statement

```
SECTIONS
{
    .text: load = EXTMEM
    UNION: run = DRAM
    {
        .pbss1:   {  file1.obj(.pbss) }
        .pbss2:   {  file2.obj(.pbss) }
    }
    .pbss3:   run =  DRAM { globals.obj(.pbss) }
```

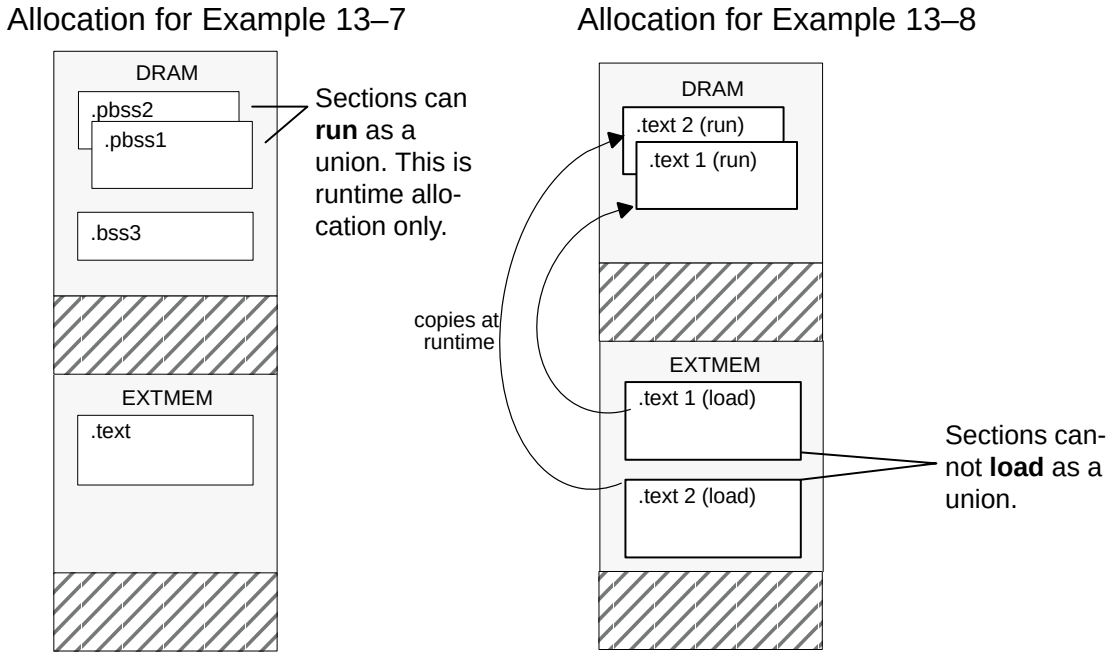
In Example 13–7, the .pbss sections from file1.obj and file2.obj are allocated *at the same address* in DRAM. The union occupies as much space in the memory map as its largest component. The components of a union remain independent sections; they are simply allocated together as a unit.

Allocation of a section as part of a union affects only its *run address*. **Under no circumstances can sections be overlaid for loading**. If an initialized section is a union member (an initialized section has raw data, such as .text), its load allocation **must** be separately specified. For example:

Example 13–8. Separate Load Addresses for UNION Sections

```
    UNION: run = DRAM
    {
        .text1: load = EXTMEM, {   file1.obj(.text) }
        .text2: load = EXTMEM, {   file2.obj(.text) }
    }
```

Figure 13–5. Memory Allocation Defined by Example 13–7 and Example 13–8



Since the .text sections contain data, they cannot *load* as a union, although they can be *run* as a union. Therefore, each requires its own load address. If you fail to provide a load allocation for an initialized section within a union, the linker issues a warning and allocates load space anywhere that space permits in configured memory.

Uninitialized sections do not require load addresses. Uninitialized sections are not loaded.

The UNION statement applies only to allocation of run addresses, so it is unnecessary to specify a load address for the union itself. For purposes of allocation, the union is treated as an uninitialized section: any one allocation specified is considered a run address, and, if both are specified, the linker issues a warning and ignores the load address.

13.9.2 Grouping Output Sections Together

The SECTIONS directive has a GROUP option that forces several output sections to be allocated contiguously. For example, assume that a section named *term_rec* contains a termination record for a table in the .data section. You can force the linker to allocate *.data* and *term_rec* together:

```
SECTIONS
{
   .text               /* Normal output section         */
   .bss                /* Normal output section         */
   GROUP 0x03000000h : /* Specify a group of sections   */
   {
     .data             /* First section in the group    */
     term_rec          /* Allocated immediately after   */
                       /* .data                         */
   }
}
```

You can use binding, alignment, or named memory to allocate a GROUP in the same way a single output section is allocated. In the preceding example, the GROUP is bound to address 0x03000000. This means that .data is allocated at 0x03000000, and term_rec follows it in memory.

Note: You Cannot Specify Addresses for Sections Within a GROUP

When you use the GROUP option, binding, or allocation into named memory can be specified *for the group only.* You cannot use binding, or named memory for sections *within* a group.

13.10 Default Allocation Algorithm

The MEMORY and SECTIONS directives provide flexible methods for building, combining, and allocating sections. However, any memory locations or sections that you choose *not* to specify must still be handled by the linker. The linker uses default algorithms to build and allocate sections within the specifications you supply. Subsections 13.10.1, below and 13.10.2, *General Rules for Output Sections*, describe default allocation algorithms.

13.10.1 Allocation Algorithm

If you do not use the MEMORY and SECTIONS directives, the linker allocates output sections as though the following definitions are specified.

❑ **Default allocation for TMS320C8x devices**

```
MEMORY
{
    DRAMS    : origin = 0x0004    , length = 0x0ffc
    DRAM2    : origin = 0x8000    , length = 0x0800
    PRAM     : origin = 0x01000200, length = 0x0600
    EXTMEM   : origin = 0x02000000, length = 0x80000
}

SECTIONS
{
    .text    : ALIGN(16) {} > EXTMEM
    .ptext   : ALIGN(16) {} > EXTMEM
    .bss     : ALIGN(4 ) {} > EXTMEM
    .data    : ALIGN(4 ) {} > EXTMEM
    .const   : ALIGN(4 ) {} > EXTMEM       /* only w/ –c, –cr, –pc options */
    .switch  : ALIGN(4 ) {} > EXTMEM       /* only w/ –c, –cr, –pc options */
    .sysmem  : ALIGN(4 ) {} > EXTMEM       /* only w/ –c, –cr, –pc options */
    .stack   : ALIGN(4 ) {} > EXTMEM       /* only w/ –c, –cr, –pc options */
    .cinit   : ALIGN(4 ) {} > EXTMEM       /* only w/ –c, –cr, –pc options */
    .pcinit  : ALIGN(4 ) {} > EXTMEM       /* only w/ –pc option           */
    .pbss    : ALIGN(4 ) {} (PASS) > DRAMS
    .psysmem : ALIGN(4 ) {} (PASS) > DRAM2 /* only w/ –pc option           */
    .pstack  : ALIGN(4 ) {} (PASS) > PRAM  /* only w/ –pc option           */
}
```

All .text input sections are concatenated to form a .text output section in the executable output file, and all .data input sections are combined to form a .data output section, etc.

Note that if you use a MEMORY directive in a linker command file, no part of the default memory specification is defined. Any memory not specifically allocated in a MEMORY directive is unconfigured, and unusable by the linker. Similarly, if you use a SECTIONS directive, no part of the default SECTIONS allocation is performed. Allocation is performed according to the rules specified by your SECTIONS directive and the general algorithm described in subsection 13.10.2, *General Rules for Output Sections*.

13.10.2 General Rules for Output Sections

An output section can be formed in one of two ways:

- As the result of a SECTIONS directive definition.
- By combining input sections with the same names into output sections that are not defined in a SECTIONS directive.

If an output section is formed as a result of a SECTIONS directive, this definition completely determines the section's contents. (See Section 13.7, *The SECTIONS Directive*, for examples of how to define an output section's content.)

An output section can also be formed when input sections are encountered that are not specified by any SECTIONS directive. In this case, the linker combines all such input sections that have the same name into an output section with this name. For example, suppose the files f1.obj and f2.obj both contain named sections called *Vectors* and that the SECTIONS directive does not define an output section to contain them. The linker combines the two *Vectors* sections from the input files into a single output section named *Vectors*, allocates it into memory, and includes it in the output file.

After the linker determines the composition of all the output sections, it must allocate them into configured memory. The MEMORY directive specifies which portions of memory are configured; if there is no MEMORY directive, the linker uses the default configuration.

The linker's allocation algorithm attempts to minimize memory fragmentation. This allows memory to be used more efficiently and increases the probability that your program will fit into memory. This is the algorithm:

1) Output sections for which you have supplied a specific binding address are placed in memory at that address.
2) Output sections that are included in a specific, named memory range or that have memory attribute restrictions are allocated. Each output section is placed into the first available space within the named area, considering alignment where necessary.
3) Any remaining sections are considered for allocation in the order in which they are defined. Sections not defined in a SECTIONS directive are also considered for allocation in the order in which they are encountered. Each remaining output section is placed into the first available memory space, considering alignment where necessary, but without regard for the type of memory.

13.11 Special Section Types (DSECT, COPY, NOLOAD, and PASS)

You can assign four special types to output sections: DSECT, COPY, NOLOAD, and PASS. These types affect the way that the section is treated when it is linked and loaded. You can assign a type to a section by placing the type (enclosed in parentheses) after the section definition. For example:

```
SECTIONS
{
   sec1 : {f1.obj} (DSECT)  0x200
   sec2 : {f2.obj} (COPY)   0x400
   sec3 : {f3.obj} (NOLOAD) 0x600
   sec4 : {f4.obj} (PASS)   0x800
}
```

- The **DSECT** type creates a dummy section that has the following qualities:
  - It is not included in the output section memory allocation. It takes up no memory and is not included in the memory map listing.
  - It can overlay other output sections, other DSECTs, and unconfigured memory.
  - Global symbols defined in a dummy section are relocated normally. They appear in the output module's symbol table with the same value they would have if the DSECT had actually been loaded. These symbols can be referenced by other input sections.
  - Undefined external symbols found in a DSECT cause specified archive libraries to be searched.
  - The section's contents, relocation information, and line number information are not placed in the output module.

  In the preceding example, none of the sections from f1.obj are allocated, but all the symbols are relocated as though the sections were linked at address 0x200. The other sections can refer to any of the global symbols in sec1.
- A **COPY** section is similar to a DSECT section, except that its contents and associated information are written to the output module. The .cinit section that contains initialization tables for the TMS320C8x MP C compiler has this attribute under the RAM model.
- A **NOLOAD** section differs from a normal output section in one respect: the section's contents, relocation information, and line number information are not placed in the output module. The linker allocates space for it, it appears in the memory map listing, etc.

- The **PASS** keyword is used to define a PASS output section. When a PASS output section is defined it is identical in all ways to a normal output section except that the PASS attribute is attached to it.

  This attribute is useful when an output section of one link step is used as an input section in another link step. When the linker sees an input section that has the PASS attribute, it will:

  - Not group or concatenate it with any other section.
  - Will not change the address that was previously assigned to the section.
  - Will not prevent other sections from being allocated into the same address space.

  In other words, it is simply passed through.

  This is very useful for doing task level linking, and allowing uninitialized sections to overlay each other. A PASS section is only useful when partial linking or task level linking is used.

  In the preceding example, all of the input sections from f4.obj are concatenated into one ouput section sec4 which is allocated at address 0x800. The PASS attribute is attached to this output section. If the section sec4 is used as an input section to another link step, then the PASS attribute will inform that link step to simply pass sec4 through to the output, without changing its allocation, reserving space for it, or grouping it with any other sections.

  Refer to the subsection 13.3.15, *Task Level Linking (–t option)* or Chapter 14, *Linking PP and MP Files: An Extended Example,* for more information on task level linking.

13.12 Assigning Symbols at Link Time

Linker assignment statements allow you to define external (global) symbols and assign values to them at link time. You can use this feature to initialize a variable or pointer to an allocation-dependent value.

13.12.1 Syntax of Assignment Statements

The syntax of assignment statements in the linker is similar to that of assignment statements in the C language:

symbol	**=**	*expression;*	assigns expression to symbol
symbol	**+ =**	*expression;*	adds expression to symbol
symbol	**– =**	*expression;*	subtracts expression from symbol
symbol	*** =**	*expression;*	multiplies symbol by expression
symbol	**/ =**	*expression;*	divides symbol by expression

The symbol should be defined externally in the program. If it is not, the linker defines a new symbol and enters it into the symbol table. The expression must follow the rules defined in subsection 13.12.3, *Assignment Expressions*. Assignment statements must be terminated with a semicolon.

The linker processes assignment statements *after* it allocates all the output sections. Thus, if an expression contains a symbol, the address used for that symbol reflects the symbol's address in the executable output file.

For example, suppose a program reads data from one of two tables identified by two external symbols, Table1 and Table2. The program uses the symbol cur_tab as the address of the current table. cur_tab must point to either Table1 or Table2. You could accomplish this in the assembly code, but you would need to reassemble the program in order to change tables. Instead, you can use a linker assignment statement to assign cur_tab at link time:

```
prog.obj           /* Input file                          */
cur_tab = Table1;  /* Assign cur_tab to one of the tables  */
```

13.12.2 Assigning the SPC to a Symbol

A special symbol, denoted by a dot (.), represents the current value of the SPC during allocation. The linker's "." symbol is analogous to the assembler's $ symbol. The "." symbol can be used only in assignment statements within a SECTIONS directive because "." is meaningful only during allocation, and SECTIONS controls the allocation process.

Note that this symbol refers to the current run address, not the current load address, of the section.

For example, suppose a program needs to know the address of the beginning of the .data section. By using the .global or .system directive, you can create an external undefined variable called Dstart in the program. Then assign the value of "." to Dstart:

```
SECTIONS
{
   .text:
   .data:   { Dstart = .; }
   .bss:
}
```

This defines Dstart to be the ultimate linked address of the .data section. (Dstart is assigned *before* .data is allocated.) The linker will relocate all references to Dstart.

A special type of assignment assigns a value to the "." symbol. This adjusts the location counter within an output section and creates a hole between two input sections. Any value assigned to "." to create a hole is relative to the beginning of the section, not to the address actually represented by ".". Assignments to "." and holes are described in Section 13.13, *Creating and Filling Holes*.

13.12.3 Assignment Expressions

These rules apply to linker expressions:

- Expressions can contain global symbols, constants, and the C language operators listed in Table 13–2.
- All numbers are treated as 32-bit integers.
- Constants are identified by the linker in the same manner as they are by the assembler. That is, numbers are recognized as decimal unless they have a suffix (H or h for hexadecimal and Q or q for octal). C language prefixes are also recognized (0 for octal and 0x for hex). Hexadecimal constants must begin with a digit if the h suffix is used. No binary constants are allowed.
- Symbols within an expression have only the value of the symbol's *address.* No type-checking is performed.
- Linker expressions can be absolute or relocatable. If an expression contains **any** relocatable symbols (and zero or more constants or absolute symbols), it is relocatable. Otherwise, the expression is absolute. If a symbol is assigned the value of a relocatable expression, the symbol is relocatable; if it is assigned the value of an absolute expression, the symbol is absolute.

The linker supports the C language operators listed in Table 13–2 in order of precedence. Operators in the same group have the same precedence. Besides the operators listed in Table 13–2, the linker also has an *align* operator that allows a symbol to be aligned on an *n*-byte boundary within an output section (*n* is a power of 2). For example, the expression

```
. = align(16);
```

aligns the SPC within the current section on the next 16-byte boundary. Because the align operator is a function of the current SPC, it can be used only in the same context as "." —that is, within a SECTIONS directive.

Table 13–2. Operators in Assignment Expressions

Group 1 (Highest Precedence)		**Group 6**	
!	Logical not	&	Bitwise AND
~	Bitwise not		
–	Negative		
Group 2		**Group 7**	
*	Multiplication	|	Bitwise OR
/	Division		
%	Mod		
Group 3		**Group 8**	
+	Addition	&&	Logical AND
–	Minus		
Group 4		**Group 9**	
>>	Arithmetic right shift	||	Logical OR
<<	Arithmetic left shift		
Group 5		**Group 10 (Lowest Precedence)**	
==	Equal to	=	Assignment
! =	Not equal to	+ =	A + = B
>	Greater than	– =	→
<	Less than	* =	A = A + B
< =	Less than or equal to	/ =	A – = B
> =	Greater than or equal to		→

13.12.4 Symbols Defined by the Linker

The linker automatically defines several symbols that a program can use at runtime to determine where a section is linked. These symbols are external, so they appear in the link map. They can be accessed in any assembly language module if they are declared with a .global directive. Values are assigned to these symbols as follows:

.text	is assigned the first address of the .text output section. (It marks the *beginning* of MP executable code.)
etext	is assigned the first address following the .text output section. (It marks the *end* of MP executable code.)
.data	is assigned the first address of the .data output section. (It marks the *beginning* of initialized data tables.)
.edata	is assigned the first address following the .data output section. (It marks the *end* of initialized data tables.)
.bss	is assigned the first address of the .bss output section. (It marks the *beginning* of MP uninitialized data.)
end	is assigned the first address following the .bss output section. (It marks the *end* of MP uninitialized data.)
cinit	is assigned the first address in the .cinit section if the –c or –cr options are used.
pcinit	is assigned the first address in the .pcinit section if the –pc option is used.
__STACK_SIZE	is assigned the size of the .stack section.
$_STACK_SIZE	is assigned the size of the .pstack section.
__SYSMEM_SIZE	is assigned the size of the .sysmem section.
$_SYSMEM_SIZE	is assigned the size of the .psysmem section.

13.13 Creating and Filling Holes

The linker provides you with the ability to create areas *within output sections* that have nothing linked into them. These areas are called **holes**. In special cases, uninitialized sections can also be treated as holes. This section describes how the linker handles such holes and how you can fill holes (and uninitialized sections) with a value.

13.13.1 Initialized and Uninitialized Sections

An output section contains either:

- Raw data for the *entire* section, **or**
- *No* raw data.

A section that has raw data is referred to as **initialized.** This means that the object file contains the actual memory image contents of the section. When the section is loaded, this image is loaded into memory at the section's specified starting address. The .text and .data sections **always** have raw data if anything was assembled into them. Named sections defined with the .sect assembler directive also has raw data.

By default, the .bss section and sections defined with the .usect directive have no raw data (they are **uninitialized**). They occupy space in the memory map but have no actual contents. Uninitialized sections typically reserve space in RAM for variables. In the object file, an uninitialized section has a normal section header and may have symbols defined in it; however, no memory image is stored in the section.

13.13.2 Creating Holes

You can create a hole in an initialized output section. A hole is created when you force the linker to leave extra space between input sections within an output section. When such a hole is created, *the linker must supply raw data for the hole.*

Holes can be created only *within* output sections. There can also be spaces *between* output sections, but such spaces are not holes. There is no way to fill or initialize the space between output sections.

To create a hole in an output section, you must use a special type of linker assignment statement within an output section definition. The assignment statement modifies the SPC (denoted by ".") by adding to it, assigning a greater value to it, or aligning it on an address boundary. The operators, expressions, and syntaxes of assignment statements are described in Section 13.12, *Assigning Symbols at Link Time*.

The following example uses assignment statements to create holes in output sections:

```
SECTIONS
{
   outsect:
   {
      file1.obj(.text)
      . += 100h;  /* Create a hole with size 100h   */
      file2.obj(.text)
      . = align(16);/*Create a hole to align the SPC */
      file3.obj(.text)
   }
}
```

The output section outsect is built as follows:

1) The .text section from file1.obj is linked in.
2) The linker creates a 256-byte hole.
3) The .text section from file2.obj is linked in after the hole.
4) The linker creates another hole by aligning the SPC on a 16-byte boundary.
5) Finally, the .text section from file3.obj is linked in.

All values assigned to the “.” symbol within a section refer to the *relative address within the section.* The linker handles assignments to the “.” symbol as if the section started at address 0 (even if you have specified a binding address). Consider the statement . = align(16) in the example. This statement effectively aligns file3.obj .text to start on a 16-byte boundary within outsect. If outsect is ultimately allocated to start on an address that is not aligned, file3.obj .text will not be aligned either.

Note that the “.” symbol refers to the current run address, not the current load address, of the section.

Expressions that decrement “.” are illegal. For example, it is invalid to use the –= operator in an assignment to “.”. The most common operators used in assignments to “.” are += and align.

If an output section contains all input sections of a certain type (such as .text), you can use the following statements to create a hole at the beginning or end of the output section:

```
.text:    {  .+= 100h; }     /* Hole at the beginning */
.data:    {
           *  (.data)
           += 100h;
           }                 /* Hole at the end       */
```

Another way to create a hole in an output section is to combine an uninitialized section with an initialized section to form a single output section. *In this case, the linker treats the uninitialized section as a hole and supplies data for it.* Here is an example of creating a hole in this way:

```
SECTIONS
{
   outsect:
   {
      file1.obj(.text)
      file1.obj(.bss)    /* This becomes a hole */
   }
}
```

Because the .text section has raw data, all of outsect must also contain raw data. Therefore, the uninitialized .bss section becomes a hole.

Note that uninitialized sections become holes only when they are combined with initialized sections. If several uninitialized sections are linked together, the resulting output section is also uninitialized.

13.13.3 Filling Holes

Whenever there is a hole in an initialized output section, the linker must supply raw data to fill it. The linker fills holes with a 32-bit fill value that is replicated through memory until it fills the hole. The linker determines the fill value as follows:

1) If the hole is formed by combining an uninitialized section with an initialized section, you can specify a fill value for the uninitialized section. Follow the section name with an = sign and a 32-bit constant:

```
SECTIONS
{
   outsect:
   {
   file1.obj(.text)
   file2.obj(.bss) = 0xFFFFFFFF/* Fill this hole   */
   }                           /* with 0xFFFFFFFF  */
}
```

2) You can also specify a fill value for all the holes in an output section by supplying the fill value after the section definition:

```
SECTIONS
{
   outsect:fill=0xFFFFFFFF /* fills holes with       */
                           /* with 0xFFFFFFFF        */
   {
      . += 10h;            /* This creates a hole    */
      file1.obj(.text)
      file1.obj(.bss)      /*This creates another hole */
   }
}
```

3) If you do not specify an initialization value for a hole, the linker fills the hole with the value specified with –f. For example, suppose the command file link.cmd contains the following SECTIONS directive:

```
SECTIONS
{
   .text: { .= 100; }    /* Create a 100-byte hole */
}
```

Now invoke the linker with the –f option:

```
mvplnk –f 0xFFFFFFFF link.cmd
```

This fills the hole with FFFFFFFFh.

4) If you do not specify a fill value, the linker fills holes with 0s.

Whenever a hole is created and filled in an initialized output section, the hole is identified in the link map along with the value the linker uses to fill it.

13.13.4 Explicit Initialization of Uninitialized Sections

An uninitialized section becomes a hole only when it is combined with an initialized section. When uninitialized sections are combined with each other, the resulting output section remains uninitialized and has no raw data in the output file.

However, you can force the linker to initialize an uninitialized section by specifying an explicit fill value for it in the SECTIONS directive. This causes the entire section to have raw data (the fill value). For example:

```
SECTIONS
{
   .bss: fill = 0x12345678  /* Fills .bss with 12345678 */
}
```

Note: Filling Sections

Because filling a section (even with 0s) causes raw data to be generated for the entire section in the output file, your output file will be very large if you specify fill values for large uninitialized sections or holes.

13.14 Partial (Incremental) Linking

An output file that has been linked can be linked again with additional modules. This is known as **partial linking** or incremental linking. Partial linking allows you to partition large applications, link each part separately, and then link all the parts together to create the final executable program.

Follow these guidelines for producing a file that you will relink:

- Intermediate files **must** have relocation information. Use the –r option when you link the file the first time.
- Intermediate files **must** have symbolic information. By default, the linker retains symbolic information in its output. Do not use the –s option if you plan to relink a file, because –s strips symbolic information from the output module.
- Intermediate link steps should be concerned only with the formation of output sections and not with allocation unless you define PASS sections. All allocation and binding should be performed in the final link step.

The following example shows how you can use partial linking:

Step 1: Link the file file1.com; use the –r option to retain relocation information in the output file tempout1.out.

```
mvplnk –r –o tempout1 file1.com
```

> **Note: See Also Task Level Linking**
>
> Task level linking is a variation of partial linking. For more information about task level linking, see subsection 13.3.15, *Task Level Linking (–t Option)*.

Step 2: Link the file file2.com; use the –r option to retain relocation information in the output file tempout2.out.

```
mvplnk –r –o tempout2 file2.com
```

Step 3: Link tempout1.out and tempout2.out:

```
mvplnk –m final.map –o final.out tempout1.out tempout2.out
```

13.15 Linking C Code

The TMS320C8x MP and PP optimizing C compilers produce assembly language source code that can be assembled and linked. For example, a C program consisting of modules prog1 prog2, can be compiled, assembled, and then linked linked to produce an executable file call prog.out:

```
mvplnk –c –o prog.out prog1.obj prog2.obj mp_rts.lib
```

This example assumes that we are linking code generated by the MP C compiler. The –c option tells the linker to use special conventions that are defined by the MP C environment. The archive library mp_rts.lib, contains runtime support functions.

If the above example linked code generated by the PP C compiler, then we would have used the –pc option instead of the –c option, and we would have linked with the library pp_rts.lib instead of mp_rts.lib. The –pc option tells the linker to use special conventions that are defined by the PP C environment.

For more information about C, including the runtime environment and runtime support functions, see Chapter 3, *PP Runtime Environment*, Chapter 4, *MP Runtime Environment,* and Chapter 5, *Runtime-Support Functions*. For more information about linking MP and PP code together, see Chapter 14, *Linking PP and MP Files: An Extended Example*.

13.15.1 Runtime Initialization

To ensure proper execution, MP C programs are linked with an object module boot.obj out of mp_rts.lib, and PP C programs are linked with an object module boot.obj out of pp_rts.lib. These boot routines define C startup code in a routine called c_int00. The function c_int00 calls the function main to begin execution of your C code. In addition to calling your C code, the c_int00 routines also contain code and data for initializing the runtime environment. The module performs the following tasks:

- It sets up the stack
- It sets up any configuration registers that ensure proper execution.
- It processes the runtime initialization table and autoinitializes global variables (in the ROM model).
- It calls main.
- It calls the ANSI defined exit routine.

The –c or –cr linker options will automatically indicate that the MP version of boot.obj should be used in the current link. The –pc option will automatically indicate that the PP version of boot.obj should be used in the current link. The above options just indicate that boot.obj is needed, you still need to specify any required libraries to the linker with the –l option.

In certain circumstances, especially in PP C code, it may be possible to bypass the C startup routine c_int00. The list below summarizes these circumstances:

- A call to exit is not needed after the execution of your main routine.
- No global or static data needs to be initialized.
- Your main routine will perform any stack setup as needed.
- Your main routine will initialize configuration registers as needed. For example resetting the lctl register on the PP, or enabling floating point operations on the MP with the IE register.

13.15.2 Setting the Size of the MP and PP Stack and Heap Sections

MP C uses two uninitialized sections called .sysmem and .stack for the memory pool used by the malloc() functions and the runtime stack respectively. PP C also uses two uninitialized section called .psysmem and .pstack for the same purposes. You can set the size of these section by using the –heap, –stack, –pheap, or –pstack linker options and specifying the size in bytes of the section as a constant immediately after the option. The default size for the MP .sysmem and .stack sections is 1K bytes. The default size for the PP .psysmem and .pstack sections is 128 bytes. For more information on the –heap, –stack, –pheap, or –pstack options. see Section 13.3, *Linker Options*.

13.15.3 Autoinitialization

The MP and PP C compilers produce tables of data for autoinitializing global variables. The MP tables are in a section called .cinit. The PP tables are in a section called .pcinit. Autoinitialization of MP global variables can occur in either a RAM model or a ROM model. For more information about the MP autoinitialization table format and the differences between the RAM and ROM models, see Section 4.8, *System Initialization*. Autoinitialization of PP global variables can occur only in a ROM model. For more information about the PP autoinitialization table format, see Section 3.8, *System Initialization*.

13.15.4 The –c, –cr and –pc Linker Options

The **–c** linker option is used to specify ROM model autoinitialization for MP C code. The following list outlines what happens when you invoke the linker with the –c option.

- The symbol _c_int00 is defined as the program entry point. _c_int00 is the start of the MP's C startup code in member boot.obj of library mp_rts.lib. Defining _c_int00 as the entry point will cause the linker to look for and link in any module that defines _c_int00.
- The .cinit section is padded with a termination record so that the MP C startup code knows when to stop reading initialization tables.
- The linker defines the symbol cinit as the starting address of the .cinit section. The MP C startup routine uses this symbol as the starting point for autoinitialization.

The **–cr** linker option is used to specify RAM model autoinitialization for MP C code. The following list outlines what happens when you invoke the linker with the –cr option.

- The symbol _c_int00 is defined as the program entry point. _c_int00 is the start of the MP's C startup code in member boot.obj of library mp_rts.lib. Defining _c_int00 as the entry point will cause the linker to look for and link in any module that defines _c_int00.
- The .cinit section is padded with a termination record so that the loader knows when to stop reading initialization tables.
- The linker defines the symbol cinit as –1. This indicates that the initialization tables are not in memory, so no initialization is performed at runtime.
- The STYP_COPY flag (010h) is set in the .cinit section header. STYP_COPY is the special attribute that tells the loader to perform autoinitialization directly and not to load the .cinit section into memory. The linker does not allocate space in memory for the .cinit section.

The **–pc** linker option is used to specify ROM model autoinitialization for PP C code. The following list outlines what happens when you invoke the linker with the –pc option.

- The symbol $c_int00 is defined as the program entry point. $c_int00 is the start of the PP's C startup code in member boot.obj of library pp_rts.lib. Defining $c_int00 as the entry point will cause the linker to look for and link in any module that defines $c_int00.
- The .pcinit section is padded with a termination record so that the PP C startup code knows when to stop reading initialization tables.
- The linker defines the symbol $pcinit as the starting address of the .pcinit section. The PP C startup routine uses this symbol as the starting point for autoinitialization.

Chapter 14

Linking PP and MP Files: An Extended Example

This chapter explores the issues involved in writing, compiling, and linking C code for the PP and MP processors that will interface. The discussion will center around a simple example. The example code is interspersed with the discussion. Full, uninterrupted text of the C code and command files used in the example can be found in Appendix B, *The Linker Example Code*.

Topics

14.1 About This Example

This example computes a table of function results for the values 1 to 9 inclusive. The functions in the table are:

- **factorial(x)**, which computes the factorial of x,
- **summation(x)**, which computes the sum of the integers from 1 to x inclusive,
- **fibonacci(x)**, which computes the xth item of the fibonacci sequence, and
- **pow4(x)**, which computes x raised to the fourth power.

This example uses one PP per function to compute the results for the table. The MP will set up the arguments for the PPs, initiate the PPs, and print the results of the PPs calculations.

The example in this chapter is useful for demonstrating some of the problems that may occur when linking MP and PP C code, and for illustrating techniques for resolving those problems. The example is not meant to be a suggestion of coding style, nor is it an example of the most efficient use of the MVP with C code. Some of the coding is purposefully inefficient, simply because it better illustrates the principles involved.

The MP and PPs can be used together in a number of different ways, however, this example is based on PP tasks as separate stand-alone programs. These PP tasks can be compiled, assembled, and linked into a COFF file as if they were complete, executable files. After the PP tasks are separately linked into COFF out files, the MP code is compiled, assembled, and linked with the out files created by the individual PP link steps. The results of this final link step is an TMS320C8x executable COFF file that can be loaded and run.

The example code will be fragmented and displayed throughout the discussion as it is appropriate. The full, uninterrupted text of the example can be found in Appendix B, *The Linker Example Code*.

14.2 The MP Example Code

This section describes the code for the MP's mp.c program that sets up and launches the PPs, gathers their data, and writes the table.

Start by including header files

The MP C code starts by including two header files: stdio.h and mvp.h (see the code segment below). If your program uses printf or any other I/O routines that accept a variable number of arguments, then you must include stdio.h so that the prototypes of these functions are visible to the calling program. The mvp.h include file defines some TMS320C8x specific symbols, functions, and macros. For more information on the mvp.h and stdio.h header files, see Chapter 5, *Runtime-Support Functions*.

```
/*************************************************************************/
/* MP.C                                                                  */
/*                                                                       */
/*    MAIN() – This routine will calculate a table of function results.  */
/*             This routine will calculate the results, by initiating    */
/*             one function per PP.  The MP will wait for the results,   */
/*             and will print them in a table.                           */
/*                                                                       */
/*************************************************************************/
#include <stdio.h>
#include <mvp.h>
```

Define macros to control the PPs

After the include files, the MP C code defines macros that are used to control the reset and unhalt actions for the PPs. For more information on interrupts, PP resets, and unhalting the PPs, see The *MVP Parallel Processor User's Guide*, Chapter 3, *PP Instruction-Cache Operation and Interprocessor Communications*.

```
/*************************************************************************/
/* Define PP control macros                                              */
/*************************************************************************/
#define INTER_PPS      0x000F0000
#define RESET_PPS          asm(”\tcmnd 0x8000000F”)
#define START_PPS          asm(”\tcmnd 0x3000000F”)
#define PPSTART_VEC(pp)    *(int **)(0x010001b8 + ((pp) << 12))
#define TABLE_MAX          10
```

Use the DATA_SECTION pragma to set up .bypass section

The MP C program then uses the DATA_SECTION pragma to indicate that all variables that will be used to communicate with the PPs will be allocated in a section called *.bypass*. These variables are placed in their own section in order to avoid having them allocated to the same cache subblock as other variables.

Since the NOCACHE_xxx macros are used to reference the communication variables, they are accessed directly from external memory, and not through the cache mechanism. Mixing variables that are accessed normally with those that bypass cache may result in lost data. For more information on the NOCACHE_xxx macros and the dangers of mixing normal and cache bypass variables in the same cache subblock, see Section 4.2, *Maintaining Data Cache Coherency*. Why the NOCACHE_xxx macros are used to access the communication variables is discussed later in the example.

```
/*************************************************************************/
/* Allocate space for the communication buffers in their own section.  This */
/* will help avoid cache errors.                                           */
/*************************************************************************/
#pragma DATA_SECTION(comm_fact_num, ".bypass");
#pragma DATA_SECTION(comm_sum_num,  ".bypass");
#pragma DATA_SECTION(comm_fib_num,  ".bypass");
#pragma DATA_SECTION(comm_pow4_num, ".bypass");
#pragma DATA_SECTION(comm_fact,     ".bypass");
#pragma DATA_SECTION(comm_sum,      ".bypass");
#pragma DATA_SECTION(comm_fib,      ".bypass");
#pragma DATA_SECTION(comm_pow4,     ".bypass");
```

Define the symbols used to communicate with the PPs

Following the pragma directives, the code defines the symbols that will be used to communicate with the PPs. An array of 10 integers and a scalar are defined for each PP task. The MP will fill the arrays with the arguments to the PP functions. The PPs will read their appropriate array, compute their function on each argument, and write the results back into the array. The scalar values are used to indicate to the PPs how many arguments are in the arrays.

Note the use of the shared keyword in the declaration of the argument arrays and the argument size variables. These symbols are defined and declared as shared for two reasons:

- Shared symbols can be referenced in both MP and PP C code. Recall the PP C symbols are prefixed with a dollar sign ($) and MP C symbols are prefixed with an underscore (_), therefore MP and PP C code cannot refer to the same symbol name. The shared keyword will result in the symbol being defined with both prefixes, (i.e. _symbol and $symbol will both be defined and will be equivalent).

- The shared keyword will prevent the symbol declarations in the PP C code from being hidden by the task level links. This example will use task level linking on the PP C code.

As an alternative, the example could have used the SHARED pragma instead of the shared keyword. For more information on the shared keyword and the SHARED pragma, see Section 2.7, *The shared and sharedpp Keywords*, and Section 2.8, *Pragma Directives*.

```
/*************************************************************************/
/* Define the communication variables.  They are defined shared so that  */
/* both MP and PP code can reference them.                               */
/*************************************************************************/
shared int comm_fact_num, comm_sum_num, comm_fib_num, comm_pow4_num;

shared int comm_fact[TABLE_MAX], comm_sum [TABLE_MAX],
           comm_fib [TABLE_MAX], comm_pow4[TABLE_MAX];
```

Declare the PP entry points

The MP code next declares the entry point symbols for the PP tasks. These symbols: ep_tsk_fact, ep_tsk_sum, etc. will not be defined in any C code in this example. Instead, the entry point symbols will be defined in the linker as a result of the –t *name* option, which controls task level linking. For example, the –t tsk_fact linker option will be used to task level link the factorial PP task.

A side effect of the –t *name* option is that it will create two symbols: $ep_*name* and _ep_*name* and it will equate these symbols to the entry point of the task. That is, the –t tsk_fact option will create the symbols $ep_tsk_fact and _ep_tsk_fact and will assign these symbols the value of the entry point into the factorial PP task. The symbol _ep_tsk_fact can be referenced in MP C code simply as ep_tsk_fact.

Having the –t linker option create the two entry point symbols is very helpful when you are linking together two or more PP tasks that use the standard PP startup code. Since the standard entry point label in the PP startup code is c_int00, all PP tasks that use the standard PP startup code will have a symbol c_int00 as the entry point. The –t linker option allows you to create a unique name for each PP's entry point symbol.

```
/*************************************************************************/
/* Entry points to the functions which will be run on the PPs.           */
/*************************************************************************/
extern int ep_tsk_fact, ep_tsk_sum, ep_tsk_fib, ep_tsk_pow4;
```

Write the PP task entry point to the PP entry vectors

The main routine starts by writing the PP task entry point addresses to the appropriate PP's entry vector. The PPs, when unhalted, will start execution from the address contained in its entry point vector.

```
/*************************************************************************/
/* MAIN() – Driver routine for this example.                             */
/*************************************************************************/
main()
{
    int i;

    /*********************************************************************/
    /* Set PPs entry point vectors to function they will perform.        */
    /*********************************************************************/
    PPSTART_VEC(0) = &ep_tsk_fib;
    PPSTART_VEC(1) = &ep_tsk_sum;
    PPSTART_VEC(2) = &ep_tsk_pow4;
    PPSTART_VEC(3) = &ep_tsk_fact;
```

Set up the argument arrays for the PPs

The main routine will then set up the argument arrays for the PPs. Each array element is simply filled with the index of its position in the array, therefore the example will calculate the function table results for arguments one to nine.

Note the use of the NOCACHE_INT macros when assigning to the array elements and to the scalar that indicates the number of arguments in the array. The NOCACHE_INT macro indicates to the MP compiler that

- it should store an integer to memory, and
- it should bypass the cache and write directly to external memory.

It is necessary to bypass cache in this example so that the PPs will read the correct copy of the data. If the MP did not use cache bypass writes in this situation, the MP would have written the data to a copy of the variables in cache, and unless a cache flush occurred, the PPs would read old data from external memory.

```
/*************************************************************************/
/* Set up the argument vectors for each PP task.                         */
/*************************************************************************/
for (i=1; i < TABLE_MAX; ++i)
{
   /**********************************************************************/
   /* We bypass cache, so the write goes directly to external memory     */
   /**********************************************************************/
   NOCACHE_INT(comm_fib[i])  = i;
   NOCACHE_INT(comm_sum[i])  = i;
   NOCACHE_INT(comm_pow4[i]) = i;
   NOCACHE_INT(comm_fact[i]) = i;
}

/*************************************************************************/
/* Set the globals indicating how many arguments each PP will process.   */
/*************************************************************************/
NOCACHE_INT(comm_fib_num)  = TABLE_MAX-1;
NOCACHE_INT(comm_sum_num)  = TABLE_MAX-1;
NOCACHE_INT(comm_pow4_num) = TABLE_MAX-1;
NOCACHE_INT(comm_fact_num) = TABLE_MAX-1;
```

Reset and start the PPs

The function main will then reset the PPs and clear the INTPEN register. The INTPEN register bits are cleared by writing a 1 to them, so the example clears the register by writing all ones to it. We need to clear this register, because the PPs will set bits in this register to indicate that they have finished processing their function.

Before the MP will unhalt the PPs, it loops on the PPERROR register to ensure that all the PPs received the reset command and have halted. Main will then issue the command for all PPs to flush their instruction caches and unhalt.

```
/*************************************************************************/
/* Reset the PPs to get them in a known state.                           */
/*************************************************************************/
RESET_PPS;

/*************************************************************************/
/* Clear INTPEN flag. Each PP writes 1 into INTPEN, when done            */
/*************************************************************************/
INTPEN = ~0;

/*************************************************************************/
/* Ensure all PP's were halted by the reset, then start PP's             */
/*************************************************************************/
while ((PPERROR & INTER_PPS) != INTER_PPS);
START_PPS;
```

Wait for the PPs to finish

The MP will then print the table headers and wait for the PPs to return. The MP polls the INTPEN register, until all the PPs signal their completion. Each PP has a dedicated bit in the INTPEN register that indicates that the PP sent the MP an interrupt request.

```
/*************************************************************************/
/* Print table header.                                                   */
/*************************************************************************/
printf("      i      fib      sum     pow4     fact     \n");
printf("      _      ___      ___     ____     ____     \n");
/*************************************************************************/
/* Wait for all PP's to signal that they have completed.                 */
/*************************************************************************/
while ((INTPEN & INTER_PPS) != INTER_PPS);
```

After all the PPs are done (each PP interrupt bit is set in INTPEN), then the code loops through each argument array and prints the results. Note again that the NOCACHE_INT macro is used to read the argument arrays. This is needed to ensure that the MP reads the arrays from external memory (where the PPs wrote the results), and not from a copy in cache.

The NOCACHE_xxx macros are very useful for ensuring proper communication between the MP and another processor. However, they execute slower than a normal load or store. In practice, the NOCACHE_xxx macros would probably only be used for scalars. For larger blocks of memory, it would be more efficient to write the array normally and then call the flush() MP runtime-support function to copy the cache image back to external memory. For more information on the NOCACHE_xxx macros, see Section 4.2, *Maintaining Data Cache Coherency*. For more information on the flush() MP runtime-support function, see Section 5.5, *Runtime-Support Functions*.

```
/*************************************************************************/
/* Read results generated by the PP's and print them out.                */
/*************************************************************************/
for (i=1; i < TABLE_MAX; ++i)
{
   /**********************************************************************/
   /* Note that results are read directly from memory, because the PP's*/
   /* did not write them into cache.                                     */
   /**********************************************************************/
   printf("%7d  %7d  %7d  %7d  %7d\n", i,
      NOCACHE_INT(comm_fib[i]),  NOCACHE_INT(comm_sum[i]),
      NOCACHE_INT(comm_pow4[i]), NOCACHE_INT(comm_fact[i]));
}
}
```

14.3 The PP Example Code

The four PP tasks are almost identical in structure, so only one will be described in this chapter. However, the listings for all the PP code can be found in Appendix B, *The Linker Example Code*.

The factorial task is contained in two source files: fact.c and fact_a.c. The file fact.c contains the main routine for the factorial task, and the fact_a.c file contains a function for calculating the factorial of an integer value. The factorial function in the file fact_a.c is a pure function that accepts an argument and returns a result. It does not reference any global variables, or have any concerns with system issues, therefore the discussion of the PP C code will be concerned mainly with the file fact.c, which does reference global variables and have system issue concerns.

Start by including the header files

The PP factorial task C code starts in the file fact.c with the mvp.h file being included. The mvp.h file is included only for visibility to the prototype of the function memtrans(), which is used later in the code.

```
/*************************************************************************/
/* FACT.C                                                                */
/*                                                                       */
/*    MAIN() – This main routine will read a global array, compute       */
/*             the factorial of the arrays elements, and write the results*/
/*             back to the global array.  It then will signal the MP that */
/*             it has completed.                                         */
/*                                                                       */
/*************************************************************************/
#include <mvp.h>
```

Declare all functions and variables

After the #include statement, the code declares the PP function fact() that will be called to compute the factorial of each argument.

The PP code then declares a function signal_done() with the sharedpp keyword. Recall that the sharedpp keyword simply informs the compiler to generate a .system directive for the symbol instead of a .global directive. Also recall that the only difference between a .system and a .global directive, is that symbols declared with .system will not be hidden by a task level link, but symbols declared with .global will be hidden. The reason that the signal_done function is sharedpp is that it is a 3 line PP assembly language function that can be used by all the PP tasks. By defining it as sharedpp, it can be linked in only once in the final link step. If it were not sharedpp, then it would be hidden by a task

level link and therefore each task level link would need to link its own private copy of it. Task level linking will be discussed more later in the example. For more information on the sharedpp keyword (or the SHAREDPP pragma) see Section 2.7, *The shared and sharedpp Keywords*, and Section 2.8, *Pragma Directives*. After signal_done() is declared, the communication variables are declared. These are the same communication variables that were defined in the MP C code. They are declared in the PP code with the shared keyword to match the shared keyword in the MP definition of the variables.

The last item before the start of the main routine is the definition of the array onchip. This definition does not include any shared keywords so this variable is global to the factorial task, but will not be visible to any code outside of the factorial task, because the factorial PP C code will be task level linked. Each PP task defines an identical copy of the array onchip, but since each PP task is task level linked, all these definitions will be limited to visibility within their own task. The purpose for the array onchip is to act as an on-chip location for the PP to block transfer its argument array to and from.

Note that the PP factorial task starts with a routine named main(). Normally it would not be possible to link together files that contained more than one definition of a symbol, but this example contains 5 different main routines, one for the MP and one each for the four PP tasks, and all this code will be linked together. Once again, task level linking will allow this to occur, because the task level link on each PP task will effectively hide the definition of main(), by making it static.

```
extern int           fact(int x);    /* Routine that calculates a factorial */
extern sharedpp void signal_done();  /* Routine that signals MP             */
extern shared int    comm_fact_num;  /* Global scalar indicating # of args  */
extern shared int    comm_fact[10];  /* Global Argument array               */
int                  onchip[10];     /* On–chip copy of argument array      */

main()

{

   int i;
```

Copy the array to on-chip memory

The first action taken by the PP factorial task's main routine is a call to the PP runtime-support function memtrans(). The memtrans() function is equivalent to memcpy(), except that memtrans() performs the memory copy by issuing a packet request to the TMS320C8x transfer controller. The memcpy() function loops while reading and writing one scalar value at a time. Therefore, the memtrans() function is more efficient than the memcpy() function, however the memtrans() function assumes that no other packet requests are currently queued. If it is possible that other packet requests have been queued then you should not use memtrans(). Refer to Chapter 6 for more information on the memtrans() runtime-support function.

The result of the memtrans() function call is that the communication array is copied into the array onchip. This is performed for efficiency reasons. The communication array is in external memory and PP references to external memory are very costly. The variable onchip defined in the PP code will be allocated memory in on-chip memory, which can be referenced very quickly. Therefore the code calls memtrans to essentially cache a copy of the communication array in on-chip memory.

The PP code then simply loops through each member of the onchip array, calls the fact function on each member and writes the results back into the onchip array.

```
/*********************************************************************/
/* Block copy argument array on–chip, for faster access.             */
/*********************************************************************/
memtrans(onchip, comm_fact, sizeof(comm_fact));
for (i=1; i <= comm_fact_num; ++i) onchip[i] = fact(onchip[i]);
```

Copy array back to external memory

After the results have been computed, the memtrans() function is called to copy the onchip array back into the communications array, so that the MP can read the results.

Finally the signal_done() function is called. This function will send a message interrupt to the MP, which has the result of setting a bit in the MP's INTPEN register. Each PP will set a different bit. Refer to the *MVP Parallel Processor User's Guide* for more information on interrupts.

```
/*********************************************************************/
/* Block copy argument array back to external memory.                */
/*********************************************************************/
memtrans(comm_fact, onchip, sizeof(comm_fact));
signal_done();
}
```

14.4 Compiling and Linking the Example

Now that the code examples are all created, it is time to compile them, and link them together to form an executable object module that can be run on a TMS320C8x device. If you jumped forward to this section because it looked like the one you are most interested in, please go back and read the preceding sections. The code has been set up in a special way so that it can be compiled and linked successfully in this section.

The command line commands to compile and link the example

The first command, ppcl, is used to compile and assemble all the PP C files and the PP assembly file signal.s. The –q option will prevent banner information from being printed and the –g option informs the compiler to generate symbolic debugging information for later use in a symbolic debugger.

```
ppcl –qg  fact.c fact_a.c sum.c  sum_a.c  fib.c  fib_a.c  pow4.c signal.s
```

The second command, mpcl, is used to compile the one MP C file, mp.c. Again the –q and the –g options are used to prevent banner information from being printed to the screen and to generate debugging information, respectively.

```
mpcl –qg  mp.c
```

The next four command lines are used to perform the four task level links for the PPs. They are identical except for the names being used, so only the command line for the fact task level link will be described.

The mvplnk command is used to invoke the MVP linker. The –q option is used to prevent banner information from being printed to the screen. The COFF object files that are to be included in the link are specified next. For this task level link, files fact.obj and fact_a.obj are specified.

The –lpplnk.cmd option informs the linker to search for the file pplnk.cmd in the standard search path (typically either the current directory or one specified in the environment variable C_DIR). For more information on the linker's file search algorithm, see subsection 13.3.7, *Alter the Library Search Algorithm (–i option/C_DIR)*. The file pplnk.cmd is the sample PP linker command file that is shipped with the MVP software toolset. The contents of pplnk.cmd will be discussed later in the example.

The –t tsk_fact option indicates that a task level link is to be performed, and that the task name is tsk_fact. Recall that a task level link will hide (by making them static) any global variable that is not declared with the .system assembler directive (or a shared keyword or SHARED pragma in C). Also, the –t tsk_fact option will create an entry point symbol ep_tsk_fact that can be referenced from either MP or PP C code. Recall that the MP source code for this example refers to a symbol ep_tsk_fact, which is defined by the use of –t on this command line. For more information on the –t linker option, see subsection 13.3.15, *Task Level Linking (–t Option)*.

The –o fact.out option indicates that the name of the resulting COFF output file should be fact.out. Similarly, the –m fact.map option indicates that a linker map file should be created and the name of the map file should be fact.map.

```
mvplnk –q  fact.o fact_a.o –lpplnk.cmd –t tsk_fact –o fact.out –m fact.map
mvplnk –q  sum.o  sum_a.o  –lpplnk.cmd –t tsk_sum  –o sum.out  –m sum.map
mvplnk –q  fib.o  fib_a.o  –lpplnk.cmd –t tsk_fib  –o fib.out  –m fib.map
mvplnk –q  pow4.o            –lpplnk.cmd –t tsk_pow4 –o pow4.out –m pow4.map
```

The last command line necessary to build this example uses the mvplnk command to combine the results of the four task level links with the MP code contained in mp.obj and the PP code contained in the file signal.obj. The PP code contained in signal.o is reentrant and is common to all the PP tasks, therefore it is linked in this final step instead of in the task level link steps.

After the files mp.obj, signal.o, fact.out, sum.out, fib.out, and pow4.out are specified, a file example.cmd is specified. The file example.cmd is a linker command file that contains options and section definitions specific to this example. A detailed description of the file example.cmd contents will follow later in the example. Notice that the –l option is not used before the filename: this indicates that the file exists in the current directory and that the linker need not search other directories.

The –lmplnk.cmd options tells the linker to search for the file mplnk.cmd, which is a sample MP linker command file that is shipped with the MVP toolset. The linker will use both the example.cmd and mplnk.cmd linker command files to control memory and section allocation in the final link step. The contents of the mplnk.cmd linker command file are discussed later in the example.

The –o mvp.out option indicates that the name of the resulting COFF output file should be mvp.out. Similarly, the –m mvp.map option indicates that a linker map file should be created and the name of the map file should be mvp.map.

```
mvplnk –q mp.obj signal.o fact.out sum.out fib.out pow4.out example.cmd \
     –lmplnk.cmd –o mvp.out –m mvp.map
```

The PP linker command file

This section uses the PP sample linker command file, pplnk.cmd, that is shipped with the MVP software development toolset. Another PP linker command file pplnkl.cmd is also shipped. The only difference between the two, is that pplnkl.cmd will include the little endian library instead of the big endian library.

The linker command file starts with some linker options. First the –pc option is used to

- indicate that PP C code will be linked,
- that the linker should set the entry point to $c_int00, and
- that the ROM model of initialization will be used.

The –x option informs the linker that it should research libraries if references remain unresolved after all libraries are searched. The linker will stop researching when no more references are resolved.

The –pheap 0x800 option sets the size of the PP's memory heap (for malloc, etc.) to 2K bytes. The –pstack 0x580 options sets the size of the PP's stack to 1408 bytes, which corresponds to the size of free space in the PP's parameter ram.

The –l pp_rts.lib option is used to inform the linker that the library pp_rts.lib may be used to resolve symbol references. The pp_rts.lib file is a standard runtime-support library that is shipped with the MVP software toolset. For more information on any of the options used in the pplnk.cmd linker command file, see Section 13.3, *Linker Options*.

```
–pc
–x
–pheap  0x800
–pstack 0x580
–l pp_rts.lib
```

The MEMORY directive follows the options in the pplnk.cmd command file. The MEMORY directive for the PP defines four ranges:

- DRAM01 starting at address 0x00000004, with a length of 4K – 4 bytes,
- DRAM2 starting at address 0x00008000, with a length of 2K bytes,
- PRAM0 starting at address 0x01000200, with a length of 1.5K bytes,
- EXTMEM starting at address 0x02000000, with a length of 512K bytes.

The range DRAM01 is a combination of a PP's Data RAM 0 and Data RAM 1. The range DRAM01 starts at address 4 instead of 0, because the rules of C state that address 0 is reserved for the NULL value, and does not constitute a valid address. Of course, assembly language users can configure the extra 4 bytes at address 0. It is also possible to split this range into two ranges DRAM0 and DRAM1, if your application needs more fine grained control over allocation of your data to specific PP data RAM banks.

The memory range DRAM2 defines the PP's third data RAM, which starts at address 0x8000 and has a length of 2K bytes. The memory range PRAM0 defines the usable portion of the PP's parameter ram. A PP's parameter ram actually is 2K bytes in length, but the lower 512 bytes of it are reserved for specific purposes. It is possible to define those extra 512 bytes with a memory range, but care should be taken when allocating data into that area.

The last memory range EXTMEM is a definition of off-chip external memory. In the sample pplnk.cmd file it is defined to start at address 0x02000000 and is defined as 512K bytes.

```
MEMORY
{
    DRAM01    : o=0x00000004   l = 0x00ffc
    DRAM2     : o=0x00008000   l = 0x00800
    PRAM0     : o=0x01000200   l = 0x00600
    EXTMEM    : o=0x02000000   l = 0x80000
}
```

The SECTIONS directive follows the memory directive, and is used to map sections to memory ranges. The sample linker command file pplnk.cmd maps eight sections into memory. The first four sections: .ptext, .const, .switch, and .pcinit are initialized sections that the PP C compiler uses. They are all linked into external memory. The .bss section is used by the PP compiler to allocate space for variables declared as far. This section is also allocated in external memory, however.

The remaining three sections: .pbss, .psysmem, and .pstack, are uninitialized sections used by the PP C compiler. They are linked into on-chip memory. The .pbss section must be linked into on-chip memory because the compiler uses the PP's 15-bit offset addressing mode to reference .pbss symbols, but .psysmem and .pstack could be linked elsewhere if necessary. The .pbss section is linked into the DRAM01 range in this sample command file. The .psysmem section is linked into the DRAM2 memory range, and the .pstack section is linked into parameter RAM.

The three sections .pbss, .psysmem, and .pstack are given the PASS attribute in the sample PP linker command file. The PASS attribute will be ignored unless task level linking or partial linking is used. When either task level or partial linking is used, the PASS attribute on a section will indicate to future links that the section is to be passed through without modification. For more information about the PASS attribute, see Section 13.11, *Special Section Types (DSECT, COPY, NOLOAD, and PASS)*.

The reason the PASS attribute was used in the sample linker command file is that the PP's will use the PP-relative addressing mode for referencing data in the sections .pbss, .pstack, and .psysmem. Since the PP-relative addressing mode will relocate the reference at runtime relative to the start of the PP's data or parameter RAM base address, at link time the symbols are all allocated relative to the start of PP0's data or parameter RAM base. By doing this, it is possible for the PP code to run on any PP without any compile time, assembly time, or link time modifications.

However, it does create the possibility of two or more sections linked at the same address at link time. For example, the section .pbss in the fact.out task level link is allocated to address 0x40, the section .pbss in the sum.out task level link is also allocated to address 0x40. In this example, the task fact is executed on PP 3, and the task sum is executed in the PP 1, which will result in the link time address 0x40, being relocated at runtime to 0x3040 for .pbss in the task fact, and 0x1040 for the .pbss in the sum fact.

The two .pbss sections mentioned above actually have different runtime address, but at link time they appear to have the same address. The PASS attribute provides an easy mechanism for indicating to the linker that having two sections linked at the same address is ok. It also hides details from further link steps, because they do not need to know any information about PASS sections.

```
SECTIONS
{
    .ptext   :  > EXTMEM
    .const   :  > EXTMEM
    .switch  :  > EXTMEM
    .pcinit  :  > EXTMEM
    .bss     :  > EXTMEM
    .pbss    :  (PASS) > DRAM01
    .psysmem :  (PASS) > DRAM2
    .pstack  :  (PASS) > PRAM0
}
```

The MP linker command file

This example uses the sample linker command file, mplnk.cmd, that is shipped with the MVP software development toolset. Another MP linker command file mplnkl.cmd is also shipped. The only difference between the two, is that mplnkl.cmd will include the little endian library instead of the big endian library.

The mplnk.cmd linker command file is similar to the pplnk.cmd linker command file, except for a few differences. The first difference between the two command files is the options used. The mplnk.cmd command file uses –c instead of –pc, –heap instead of –pheap, –stack instead of –pstack, and includes the MP version of the runtime-support library mp_rts.lib instead of pp_rts.lib. The different options perform the same tasks as the PP counterparts, but they are for the MP instead.

```
–c
–x
–heap  0x2000
–stack 0x2000
–l mp_rts.lib
```

The second difference is in the number of memory ranges that are defined in the mplnk.cmd command file. The MP does not define the on-chip memory ranges, because it does not allocate data into those areas. The link that uses this linker command file will have data allocated into on-chip memory ranges, but they will have been allocated in task level links and will have the PASS attribute and will simply be passed through, without the MP command file having any effect on them.

```
MEMORY
{
    EXTMEM    : o=0x02000000   l = 0x80000
}
```

The third difference is the set of sections that are mapped to memory. The MP compiler uses a different set of sections than the PP compiler, so when MP C code is compiled the sections .text, .bss, .const, .switch, .sysmem, .stack, and .cinit will need to be allocated. The sections .ptext and .pcinit are also allocated here to handle those sections from the task level links, because they were not defined as PASS.

```
SECTIONS
{
    .text    :  > EXTMEM
    .ptext   :  > EXTMEM
    .bss     :  > EXTMEM
    .const   :  > EXTMEM
    .switch  :  > EXTMEM
    .sysmem  :  > EXTMEM
    .stack   :  > EXTMEM
    .cinit   :  > EXTMEM
    .pcinit  :  > EXTMEM
}
```

Example link linker command file

The example command file is used to provide additional control of the final link. The options and sections defined in example.cmd could have been combined with the information in mplnk.cmd and one command file created, but it is easier for the purpose of illustration to use two different command files. The example.cmd command file has two purposes:

- The command file must link the C I/O routines (printf, etc.) correctly. To do this, the command file does the following:
  - Maps the .cio section into external memory. The .cio section contains a buffer used by the C I/O routines
  - Uses the –l option to link with the mp_cio.lib library. The mp_cio.lib library contains the definitions of the C I/O routines, such as printf, scanf, etc.
  - Uses the –u option to undefine the _exit label. The C I/O routines use a special version of the ANSI standard exit routine, that informs the debugger that a program has finished. The special version is contained in the library mp_cio.lib. The standard version of exit is contained in the library mp_rts.lib. The –u _exit option will cause the special version of exit in mp_cio.lib to be linked in instead of the standard version in mp_rts.lib when example.cmd is specified before mplnk.cmd on the command line.

- The command file must prevent the .bypass section from existing on the same data cache subblock as any other section.

  To accomplish this it uses the align keyword to ensure that the .bypass section is aligned to a 64 byte (data cache subblock) boundary, and uses an assignment to the SPC (.) to create a hole between the end of the .bypass section and the next 64 byte boundary. This will make the .bypass section in the final .out file always be a multiple of 64 bytes in length, and since it is aligned to a 64 byte boundary, it will never co-exist with another section on the same data cache subblock.

```
-u _exit
-l mp_cio.lib

SECTIONS
{
    .cio     :  > EXTMEM
    .bypass  : { *(.bypass) . = align(64); } align(64) > EXTMEM
}
```

14.5 The Example Linker Map Files

The fact.map file generated for the task level link of the fact PP task.

```
*************************************************
MVP COFF Linker              Version 1.05
*************************************************
Thu Dec 22 11:05:12 1994

OUTPUT FILE NAME:   <fact.out>
ENTRY POINT SYMBOL: "$c_int00"  address: 02000820

MEMORY CONFIGURATION

          name      origin     length     attributes     fill
        --------   --------   ---------   ----------   --------
        DRAM01     00000004   000003ffc      RWIX
        DRAM2      00008000   000000800      RWIX
        PRAM0      01000200   000000600      RWIX
        EXTMEM     02000000   000080000      RWIX

SECTION ALLOCATION MAP

 output                                  attributes/
section   page    origin      length       input sections
--------  ----  ----------  ----------   ----------------
.ptext     0    02000000    00000878
                  02000000    00000410     pp_rts.lib : cinit.o (.ptext)
                  02000410    00000178                : exit.o (.ptext)
                  02000588    00000140     fact.o (.ptext)
                  020006c8    000000b8     pp_rts.lib : mtrans.o (.ptext)
                  02000780    000000a0     fact_a.o (.ptext)
                  02000820    00000058     pp_rts.lib : boot.o (.ptext)

.const     0    02000000    00000000     UNINITIALIZED

.switch    0    02000000    00000000     UNINITIALIZED

.pcinit    0    02000878    00000010
                  02000878    0000000c     pp_rts.lib : exit.o (.pcinit)
                  02000884    00000004     --HOLE-- [fill = 00000000]

.bss       0    02000000    00000000     UNINITIALIZED
                  02000000    00000000     fact.o (.bss)
                  02000000    00000000     pp_rts.lib : exit.o (.bss)
                  02000000    00000000                : cinit.o (.bss)
                  02000000    00000000                : boot.o (.bss)
                  02000000    00000000                : mtrans.o (.bss)
                  02000000    00000000     fact_a.o (.bss)

.pbss      0    00000040    00000140     PASS SECTION
                  00000040    00000084     pp_rts.lib : exit.o (.pbss)
                  000000c4    00000028     fact.o (.pbss)
                  00000100    00000040     pp_rts.lib : cinit.o (.pbss)
                  00000140    00000040                : mtrans.o (.pbss)

.psysmem   0    00008000    00000000     PASS SECTION

.pstack    0    01000200    00000580     PASS SECTION
                  01000200    00000080     pp_rts.lib : boot.o (.pstack)

.text      0    00000000    00000000     UNINITIALIZED
                  00000000    00000000     fact.o (.text)
                  00000000    00000000     pp_rts.lib : exit.o (.text)
                  00000000    00000000                : cinit.o (.text)
                  00000000    00000000                : boot.o (.text)
                  00000000    00000000                : mtrans.o (.text)
                  00000000    00000000     fact_a.o (.text)
```

```
.data      0    00000000    00000000     UNINITIALIZED
                  00000000    00000000       fact.o (.data)
                  00000000    00000000       pp_rts.lib : exit.o (.data)
                  00000000    00000000                  : cinit.o (.data)
                  00000000    00000000                  : boot.o (.data)
                  00000000    00000000                  : mtrans.o (.data)
                  00000000    00000000       fact_a.o (.data)

GLOBAL SYMBOLS

address  name                                address  name
--------- ----                                --------- ----
00000580 $_STACK_SIZE                        00000000 etext
02000000 $_auto_init                         00000000 .text
01000200 $_stack                             00000000 edata
02000548 $abort                              00000000 .data
020004d8 $atexit                             000000c4 $onchip
02000820 $c_int00                            00000580 $_STACK_SIZE
UNDEFED  $comm_fact_num                      01000200 $_stack
UNDEFED  $comm_fact                          02000000 $_auto_init
02000820 $ep_tsk_fact                        02000000 end
02000410 $exit                               02000000 .bss
02000780 $fact                               02000410 $exit
02000588 $main                               020004d8 $atexit
020006c8 $memtrans                           02000548 $abort
000000c4 $onchip                             02000588 $main
02000878 $pcinit                             020006c8 $memtrans
UNDEFED  $signal_done                        02000780 $fact
02000000 .bss                                02000820 $c_int00
00000000 .data                               02000820 _ep_tsk_fact
00000000 .text                               02000820 $ep_tsk_fact
02000820 _ep_tsk_fact                        02000878 $pcinit
ffffffff cinit                               ffffffff cinit
00000000 edata                               UNDEFED  $signal_done
02000000 end                                 UNDEFED  $comm_fact
00000000 etext                               UNDEFED  $comm_fact_num

[24 symbols]
```

The mvp.map file generated by the linker for the final link of this example

```
*****************************************************
MVP COFF Linker              Version 1.05
*****************************************************
Thu Dec 22 11:05:21 1994

OUTPUT FILE NAME:   <mvp.out>
ENTRY POINT SYMBOL: ”_c_int00”  address: 02004534

SECTION ALLOCATION MAP

 output                                  attributes/
section   page    origin      length       input sections
--------  ----  ----------  ----------   ----------------
.cio       0    02000000    00000120     UNINITIALIZED
                  02000000    00000120     mp_cio.lib : trgmsg.obj (.cio)

.bypass    0    02000180    00000100     UNINITIALIZED
                  02000180    000000b0     mp.obj (.bypass)

.text      0    02000280    0000433c
                  02000280    000001e0     mp.obj (.text)
                  02000460    00000000     mp_rts.lib : ctype.obj (.text)
                  02000460    00000000                : sysmem.obj (.text)
                  02000460    00000000                : errno.obj (.text)
                  02000460    00000000     mp_cio.lib : data.obj (.text)
                  02000460    00000000     signal.o (.text)
                  02000460    00000000     pow4.out (.text)
                  02000460    00000000     fib.out (.text)
                  02000460    00000000     sum.out (.text)
                  02000460    00000000     fact.out (.text)
                  02000460    000000d4     mp_cio.lib : exit.obj (.text)
                  02000534    000005ec                : flsbuf.obj (.text)
                  02000b20    00000160                : getc.obj (.text)
                  02000c80    000005f4                : lowlev.obj (.text)
                  02001274    00000098                : printf.obj (.text)
                  0200130c    00000528                : trgdrv.obj (.text)
                  02001834    00000100                : trgmsg.obj (.text)
                  02001934    0000185c                : doprnt.obj (.text)
                  02003190    0000017c                : filbuf.obj (.text)
                  0200330c    00000210                : fwrite.obj (.text)
                  0200351c    00000240     mp_rts.lib : ecvt.obj (.text)
                  0200375c    00000254                : fcvt.obj (.text)
                  020039b0    00000020                : isdigit.obj (.text)
                  020039d0    00000020                : isupper.obj (.text)
                  020039f0    000000a8                : ltoa.obj (.text)
                  02003a98    00000068                : memchr.obj (.text)
                  02003b00    000000e8                : memcpy.obj (.text)
                  02003be8    00000814                : memory.obj (.text)
                  020043fc    0000005c                : strchr.obj (.text)
                  02004458    00000024                : strcmp.obj (.text)
                  0200447c    00000018                : strcpy.obj (.text)
                  02004494    00000044                : strlen.obj (.text)
                  020044d8    00000040                : strncpy.obj (.text)
                  02004518    0000001c                : i_mod.obj (.text)
                  02004534    00000088                : boot.obj (.text)
```

```
.bss       0    020045c0    000006b0     UNINITIALIZED
                  020045c0    00000240       mp_cio.lib : data.obj (.bss)
                  02004800    00000000       sum.out (.bss)
                  02004800    00000000       mp_cio.lib : fwrite.obj (.bss)
                  02004800    00000000                  : filbuf.obj (.bss)
                  02004800    00000000       mp_rts.lib : isupper.obj (.bss)
                  02004800    00000000                  : isdigit.obj (.bss)
                  02004800    00000000       fact.out (.bss)
                  02004800    00000000       signal.o (.bss)
                  02004800    00000000       mp_cio.lib : printf.obj (.bss)
                  02004800    00000000                  : getc.obj (.bss)
                  02004800    00000000       fib.out (.bss)
                  02004800    00000000       pow4.out (.bss)
                  02004800    00000000       mp_cio.lib : trgmsg.obj (.bss)
                  02004800    00000000       mp_rts.lib : ltoa.obj (.bss)
                  02004800    00000000                  : i_mod.obj (.bss)
                  02004800    00000000                  : strncpy.obj (.bss)
                  02004800    00000000                  : strlen.obj (.bss)
                  02004800    00000000       mp.obj (.bss)
                  02004800    00000000       mp_rts.lib : sysmem.obj (.bss)
                  02004800    00000000                  : boot.obj (.bss)
                  02004800    00000000                  : strcpy.obj (.bss)
                  02004800    00000000                  : memchr.obj (.bss)
                  02004800    00000000                  : memcpy.obj (.bss)
                  02004800    00000000                  : strcmp.obj (.bss)
                  02004800    00000000                  : strchr.obj (.bss)
                  02004800    00000101                  : ctype.obj (.bss)
                  02004904    000000d0       mp_cio.lib : trgdrv.obj (.bss)
                  020049d4    000000cc                  : lowlev.obj (.bss)
                  02004aa0    00000088                  : exit.obj (.bss)
                  02004b28    00000070                  : doprnt.obj (.bss)
                  02004b98    00000064       mp_rts.lib : fcvt.obj (.bss)
                  02004bfc    00000064                  : ecvt.obj (.bss)
                  02004c60    00000008                  : memory.obj (.bss)
                  02004c68    00000004                  : errno.obj (.bss)
                  02004c6c    00000004       mp_cio.lib : flsbuf.obj (.bss)

.const     0    02004c70    000000cc
                  02004c70    00000084       mp.obj (.const)
                  02004cf4    00000000       fib.out (.const)
                  02004cf4    00000000       sum.out (.const)
                  02004cf4    00000000       fact.out (.const)
                  02004cf4    00000000       pow4.out (.const)
                  02004cf4    00000004       --HOLE-- [fill = 00000000]
                  02004cf8    00000018       mp_rts.lib : ecvt.obj (.const)
                  02004d10    00000018                  : fcvt.obj (.const)
                  02004d28    00000014       mp_cio.lib : doprnt.obj (.const)

.switch    0    02000120    00000040
                  02000120    00000000       fact.out (.switch)
                  02000120    00000000       pow4.out (.switch)
                  02000120    00000000       fib.out (.switch)
                  02000120    00000000       sum.out (.switch)
                  02000120    00000040       mp_cio.lib : doprnt.obj (.switch)

.sysmem    0    02004d40    00002000     UNINITIALIZED
                  02004d40    00000040       mp_rts.lib : sysmem.obj (.sysmem)

.stack     0    02006d40    00002000     UNINITIALIZED
                  02006d40    00000080       mp_rts.lib : boot.obj (.stack)
```

```
.cinit     0    02008d40    000002c0
                  02008d40    0000010c     mp_rts.lib : ctype.obj (.cinit)
                  02008e4c    000000b0     mp_cio.lib : doprnt.obj (.cinit)
                  02008efc    00000058                : data.obj (.cinit)
                  02008f54    00000050                : lowlev.obj (.cinit)
                  02008fa4    0000001c     mp_rts.lib : sysmem.obj (.cinit)
                  02008fc0    00000018     mp_cio.lib : exit.obj (.cinit)
                  02008fd8    0000000c     mp_rts.lib : memory.obj (.cinit)
                  02008fe4    0000000c     mp_cio.lib : flsbuf.obj (.cinit)
                  02008ff0    0000000c     mp_rts.lib : errno.obj (.cinit)
                  02008ffc    00000004     --HOLE-- [fill = 00000000]

.data      0    00000000    00000000     UNINITIALIZED
                  00000000    00000000     mp.obj (.data)
                  00000000    00000000     mp_rts.lib : ctype.obj (.data)
                  00000000    00000000                : sysmem.obj (.data)
                  00000000    00000000                : boot.obj (.data)
                  00000000    00000000                : i_mod.obj (.data)
                  00000000    00000000                : strncpy.obj (.data)
                  00000000    00000000                : strlen.obj (.data)
                  00000000    00000000                : strcpy.obj (.data)
                  00000000    00000000                : strcmp.obj (.data)
                  00000000    00000000                : strchr.obj (.data)
                  00000000    00000000                : memory.obj (.data)
                  00000000    00000000                : memcpy.obj (.data)
                  00000000    00000000                : memchr.obj (.data)
                  00000000    00000000                : ltoa.obj (.data)
                  00000000    00000000                : isupper.obj (.data)
                  00000000    00000000                : isdigit.obj (.data)
                  00000000    00000000                : fcvt.obj (.data)
                  00000000    00000000                : errno.obj (.data)
                  00000000    00000000                : ecvt.obj (.data)
                  00000000    00000000     mp_cio.lib : fwrite.obj (.data)
                  00000000    00000000                : filbuf.obj (.data)
                  00000000    00000000                : doprnt.obj (.data)
                  00000000    00000000                : data.obj (.data)
                  00000000    00000000                : trgmsg.obj (.data)
                  00000000    00000000                : trgdrv.obj (.data)
                  00000000    00000000                : printf.obj (.data)
                  00000000    00000000                : lowlev.obj (.data)
                  00000000    00000000                : getc.obj (.data)
                  00000000    00000000                : flsbuf.obj (.data)
                  00000000    00000000                : exit.obj (.data)
                  00000000    00000000     signal.o (.data)
                  00000000    00000000     pow4.out (.data)
                  00000000    00000000     fib.out (.data)
                  00000000    00000000     sum.out (.data)
                  00000000    00000000     fact.out (.data)

.ptext     0    02009000    00003118
                  02009000    000017d0     pow4.out (.ptext)
                  0200a7d0    00000878     fact.out (.ptext)
                  0200b048    00000860     fib.out (.ptext)
                  0200b8a8    00000858     sum.out (.ptext)
                  0200c100    00000018     signal.o (.ptext)

.pcinit    0    0200c118    0000004c
                  0200c118    0000001c     pow4.out (.pcinit)
                  0200c134    00000010     fib.out (.pcinit)
                  0200c144    00000010     fact.out (.pcinit)
                  0200c154    00000010     sum.out (.pcinit)

.pbss      0    00000040    00000140     PASS SECTION
                  00000040    00000140     fact.out (.pbss)

.psysmem   0    00000000    00000000     PASS SECTION
                  00000000    00000000     fact.out (.psysmem)
```

```
.pstack    0    01000200    00000580     PASS SECTION
                  01000200    00000580       fact.out (.pstack)

.pbss      0    00000040    00000140     PASS SECTION
                  00000040    00000140       sum.out (.pbss)

.psysmem   0    00000000    00000000     PASS SECTION
                  00000000    00000000       sum.out (.psysmem)

.pstack    0    01000200    00000580     PASS SECTION
                  01000200    00000580       sum.out (.pstack)

.pbss      0    00000040    00000140     PASS SECTION
                  00000040    00000140       fib.out (.pbss)

.psysmem   0    00000000    00000000     PASS SECTION
                  00000000    00000000       fib.out (.psysmem)

.pstack    0    01000200    00000580     PASS SECTION
                  01000200    00000580       fib.out (.pstack)

.pbss      0    00000040    00000140     PASS SECTION
                  00000040    00000140       pow4.out (.pbss)

.psysmem   0    00000000    00000000     PASS SECTION
                  00000000    00000000       pow4.out (.psysmem)

.pstack    0    01000200    00000580     PASS SECTION
                  01000200    00000580       pow4.out (.pstack)

GLOBAL SYMBOLS

address  name                            address  name
--------- ----                           --------- ----
02000000 $_CIOBUF_                       00000000 edata
02000204 $comm_fact_num                  00000000 .data
020001dc $comm_fact                      00002000 __SYSMEM_SIZE
020001b0 $comm_fib_num                   00002000 __STACK_SIZE
02000184 $comm_fib                       02000000 $_CIOBUF_
02000180 $comm_pow4_num                  02000000 __CIOBUF_
02000208 $comm_pow4                      02000180 _comm_pow4_num
020001ac $comm_sum_num                   02000180 $comm_pow4_num
020001b4 $comm_sum                       02000184 _comm_fib
0200aff0 $ep_tsk_fact                    02000184 $comm_fib
0200b850 $ep_tsk_fib                     020001ac _comm_sum_num
02009ca8 $ep_tsk_pow4                    020001ac $comm_sum_num
0200c0a8 $ep_tsk_sum                     020001b0 $comm_fib_num
0200c100 $signal_done                    020001b0 _comm_fib_num
020045c0 .bss                            020001b4 _comm_sum
00000000 .data                           020001b4 $comm_sum
02000280 .text                           020001dc _comm_fact
020004bc C$$EXIT                         020001dc $comm_fact
020018b4 C$$IO$$                         02000204 $comm_fact_num
02004518 I_MOD                           02000204 _comm_fact_num
020013bc _HOSTclose                      02000208 _comm_pow4
02001584 _HOSTlseek                      02000208 $comm_pow4
0200130c _HOSTopen                       02000280 _main
0200143c _HOSTread                       02000280 .text
020016f0 _HOSTrename                     02000460 _exit
02001680 _HOSTunlink                     020004bc C$$EXIT
020014e0 _HOSTwrite                      020004c4 _atexit
02000000 __CIOBUF_                       02000514 _abort
00002000 __STACK_SIZE                    02000534 __cleanup
00002000 __SYSMEM_SIZE                   02000590 _fclose
02004704 __bufendtab                     02000690 _fflush
02004aa4 __cleanup_ptr                   02000798 __flsbuf
02000534 __cleanup                       0200091c __xflsbuf
02004800 __ctypes_                       020009b8 __wrtchk
```

```
020049d4 __device
02001a4c __doprnt
02003190 __filbuf
02000a5c __findbuf
02000798 __flsbuf
020045c0 __iob
02004700 __lastbuf
02004c64 __memory_size
02004758 __smbuf
02006d40 __stack
02004a4c __stream
02004d40 __sys_memory
02004c6c __tmpdel_ptr
020009b8 __wrtchk
0200091c __xflsbuf
02000514 _abort
02000ea8 _add_device
020004c4 _atexit
02004534 _c_int00
02003e1c _calloc
02000c24 _clearerr
02001188 _close
020001dc _comm_fact
02000204 _comm_fact_num
020001b0 _comm_fib_num
02000184 _comm_fib
02000180 _comm_pow4_num
02000208 _comm_pow4
020001b4 _comm_sum
020001ac _comm_sum_num
0200351c _ecvt
0200aff0 _ep_tsk_fact
0200b850 _ep_tsk_fib
02009ca8 _ep_tsk_pow4
0200c0a8 _ep_tsk_sum
02004c68 _errno
02000460 _exit
02000590 _fclose
0200375c _fcvt
02000c48 _feof
02000c64 _ferror
02000690 _fflush
02000d88 _finddevice
02004080 _free
0200330c _fwrite
02000bd4 _getchar
02000b20 _getc
02000e10 _getdevice
020017b0 _getenv
02000c80 _getnexfildes
020039b0 _isdigit
020039d0 _isupper
02001144 _lseek
020039f0 _ltoa
02000280 _main
02003d54 _malloc
02004204 _memalign
02003a98 _memchr
02003b00 _memcpy
02003d1c _minit
02000fec _open
020049cc _parmbuf
02001274 _printf
02000b74 _putc
02000a5c __findbuf
02000b20 _getc
02000b74 _putc
02000bd4 _getchar
02000bf8 _putchar
02000c24 _clearerr
02000c48 _feof
02000c64 _ferror
02000c80 _getnexfildes
02000cec _tabinit
02000d88 _finddevice
02000e10 _getdevice
02000ea8 _add_device
02000fac _remove_device
02000fec _open
020010bc _read
02001100 _write
02001144 _lseek
02001188 _close
020011fc _unlink
02001234 _rename
02001274 _printf
0200130c _HOSTopen
020013bc _HOSTclose
0200143c _HOSTread
020014e0 _HOSTwrite
02001584 _HOSTlseek
02001680 _HOSTunlink
020016f0 _HOSTrename
020017b0 _getenv
02001834 _writemsg
020018b4 C$$IO$$
020018c0 _readmsg
02001a4c __doprnt
02003190 __filbuf
0200330c _fwrite
0200351c _ecvt
0200375c _fcvt
020039b0 _isdigit
020039d0 _isupper
020039f0 _ltoa
02003a98 _memchr
02003b00 _memcpy
02003d1c _minit
02003d54 _malloc
02003e1c _calloc
02003e9c _realloc
02004080 _free
02004204 _memalign
020043fc _strchr
02004458 _strcmp
0200447c _strcpy
02004494 _strlen
020044d8 _strncpy
02004518 I_MOD
02004534 _c_int00
020045bc etext
020045c0 .bss
020045c0 __iob
02004700 __lastbuf
02004704 __bufendtab
02004758 __smbuf
02004800 __ctypes_
020049cc _parmbuf
```

```
02000bf8 _putchar
020018c0 _readmsg
020010bc _read
02003e9c _realloc
02000fac _remove_device
02001234 _rename
020043fc _strchr
02004458 _strcmp
0200447c _strcpy
02004494 _strlen
020044d8 _strncpy
02000cec _tabinit
020011fc _unlink
02001100 _write
02001834 _writemsg
02008d40 cinit
00000000 edata
02004c70 end
020045bc etext

[117 symbols]
```

```
020049d4 __device
02004a4c __stream
02004aa4 __cleanup_ptr
02004c64 __memory_size
02004c68 _errno
02004c6c __tmpdel_ptr
02004c70 end
02004d40 __sys_memory
02006d40 __stack
02008d40 cinit
02009ca8 _ep_tsk_pow4
02009ca8 $ep_tsk_pow4
0200aff0 $ep_tsk_fact
0200aff0 _ep_tsk_fact
0200b850 $ep_tsk_fib
0200b850 _ep_tsk_fib
0200c0a8 _ep_tsk_sum
0200c0a8 $ep_tsk_sum
0200c100 $signal_done
```

Chapter 15

Linker Error Messages

This chapter discusses three types of linker errors; they are listed alphabetically within each category. In these listings, the symbol **(...)** represents the name of an object that the linker is attempting to interact with when an error occurs.

Topics

15.1 Syntax/Command Errors

These errors are caused by incorrect use of linker directives, misuse of an input expression, or invalid options. Check the syntax of all expressions, and check the input directives for accuracy. Review the various options you are using and check for conflicts.

A

absolute symbol (...) being redefined

Description An absolute symbol cannot be redefined.

adding (...) to multiple output sections

Description The input section is mentioned twice in the SECTIONS directive.

ALIGN illegal in this context

Description Alignment of a symbol can be performed only within a SECTIONS directive.

alignment for (...) must be a power of 2

Description Section alignment must be a power of 2.

Action In hexadecimal, all powers of 2 consist of a 1, 2, 4, or 8 followed by a series of zero or more 0s.

alignment for (...) redefined

Description Only one alignment is allowed for each section.

attempt to decrement DOT

Description Statements such as **.–= value** are illegal. Assignments to **dot** can be used only to create holes.

B

binding address for (...) redefined

Description Only one binding value is allowed for each section.

blocking for (...) must be a power of 2

Description Section blocking must be a power of 2.

Action In hexadecimal, all powers of 2 consist of a 1, 2, 4, or 8 followed by a series of zero or more 0s.

blocking for (...) redefined

Description Only one blocking value is allowed for each section.

C

–c requires fill value of 0 in .cinit (... overridden)

Description C runtime conventions require the .cinit tables to be terminated with 0.

can't open *filename*

Description Specified filename cannot be opened for some reason; file doesn't exist, wrong file type, etc.

Action Check spelling, pathname, environment variables, etc.

cannot resize (...), section has initialized definition in (...)

Description An *initialized* input section named .stack or .heap exists, preventing the linker from resizing the section.

cannot specify both binding and memory area for (...)

Description The two are mutually exclusive. If you wish the code to be placed at a specific address, use binding only.

command file nesting exceeded with file (...)

Description Command file nesting is allowed up to 16 levels.

E

–e flag does not specify a legal symbol name (...)

Description The –e option requires a valid symbol name as an operand.

entry point other than _c_int00 specified

Description For –c or –cr option only. The runtime conventions of the compiler assume that _c_int00 is the one and only entry point.

entry point symbol (...) undefined

Description The symbol used with the –e option is not defined.

errors in input – (...) not built

Description Previous errors prevent the creation of an output file.

F

fill value for (...) redefined

Description Only one fill value is allowed per output section. Individual holes can be filled with different values with the section definition.

fill value redefined for memory area

Description Each memory area in a MEMORY directive can have only one fill value.

G

–g flag does not specify a valid name

Description The symbol designated by –g is undefined.

I

–i path too long (...)

Description The maximum number of characters in an –i path is 256.

illegal input character

Description There is a control character or other unrecognized character in the command file.

illegal memory attributes for (...)

Description The attributes must be some combination of R, W, I, and X.

illegal operator in expression

Action Review legal expression operators.

illegal option within SECTIONS

Description The –l (lowercase L) is the only option allowed within a SECTIONS directive.

invalid path specified with –i flag

Description The operand of the –i flag must be a valid file or pathname.

invalid value for –f flag

Description –f flag must be a constant.

invalid value for –heap flag

Description –heap flag must be a constant.

invalid value for –stack flag

Description –stack flag must be a constant.

invalid value for –v flag

Description –v flag must be a constant.

L

length redefined for memory area (...)

Description Each memory area in a MEMORY directive can have only one length.

M

–m flag does not specify a valid filename

Action Specify a valid filename to write the output map file to.

memory area for (...) redefined

Description Only one named memory allocation is allowed for each output section.

memory attributes redefined for (...)

Description Only one set of memory attributes is allowed for each output section.

memory page for (...) redefined

Description Only one page allocation is allowed for each section.

MEMORY specification ignored

Description The most likely cause of this message is a syntax error in the MEMORY directive.

missing filename on –l; use –l <filename>

Description The –l (lowercase L) option requires the use of a filename operand.

misuse of DOT symbol in assignment instruction

Description The dot symbol cannot be used in assignment statements that are outside SECTIONS directives.

N

no input files

Description The linker cannot operate without at least one input COFF file.

O

–o flag does specify a valid file name : *string*

Description The filename must follow the operating system conventions.

output file has no .bss section

Description This is a warning. This section is usually present in a COFF file, though there is no requirement for it to be present.

output file has no .data section

Description This is a warning. This section is usually present in a COFF file, though there is no requirement for it to be present.

output file has no .text section

Description This is a warning. This section is usually present in a COFF file, though there is no requirement for it to be present.

origin missing for memory area (...)

Description The origin was undefined for memory area (...)

origin redefined for memory area

Description Each area in a MEMORY directive can only have one origin.

R

–r incompatible with –s (–s ignored)

Description Since the –s option strips the relocation information, and –r requests a relocatable object file, these options are in conflict with each other.

S

section (...) not built

Description The most likely cause of this message is a syntax error in the SECTIONS directive.

semicolon required after assignment

Description There is a syntax error in the command file.

statement ignored

Description This message is caused by a syntax error in an expression.

symbol referencing errors — (...) not built

Description Previous errors prevent the creation of an output file.

symbol (...) from file (...) being redefined

Description A defined symbol cannot be redefined in an assignment statement.

syntax error

Description There is a syntax error in the command file.

T

–t flag does not specify a valid name

Description The –t option requires a valid symbol name as an operand.

task entry point sym (...) cannot be redefined

Description The –t option requires that the symbol ep_NAME not be used in any code involved in the task link.

too many arguments – use a command file

Description You are limited to ten arguments on a command line or in response to prompts.

too many –i options, 7 allowed

Action Additional search directories can be specified with a C_DIR or A_DIR environment variable.

type flags for (...) redefined

Description Only one section type is allowed per section. Note that type COPY has all of the attributes of type DSECT, so DSECT need not be specified separately.

type flags not allowed for GROUP or UNION

Description Special section types apply to individual sections only.

U

–u does not specify a legal symbol name

Description The –u option must specify a legal symbol name that exists in one of the files that you are linking.

unexpected EOF(end of file)

Description There is a syntax error in the linker command file.

undefined symbol in expression

Description An assignment statement contains an undefined symbol.

unrecognized option (...)

Action Check the list of valid options.

Z

zero or missing length for memory area (...)

Description Each memory range defined with the MEMORY directive must have a nonzero length.

15.2 Allocation Errors

These error messages appear during the allocation phase of linking. They generally appear if a section or group does not fit at a certain address or if the MEMORY and SECTIONS directives conflict in some way.

B

binding address (...) for section (...) is outside all memory on page (...)

Description Each section must fall within memory configured with the MEMORY directive.

Action If you are using a linker command file, check that the MEMORY and SECTIONS directives allow enough room to ensure that no sections are being placed in unconfigured memory.

binding address (...) for section (...) overlays (...) at (...)

Description Two sections overlap and cannot be allocated.

Action If you are using a linker command file, check that the MEMORY and SECTIONS directives allow enough room to ensure that no sections are being placed in unconfigured memory.

binding address (...) incompatible with alignment for section (...)

Description The section has an alignment requirement from an .align directive or previous link. The binding address violates this requirement.

binding address (...) incompatible with blocking for (...)

Description The section has a blocking requirement from an allocation blocking specification or a previous link. The binding address violates this requirement

C

can’t align within UNION – section (...) not aligned

Description The entire UNION is treated as one unit, so the UNION can be aligned or bound to an address, but the sections making up the UNION cannot be handled individually.

can’t allocate (...), size ... (page ...)

Description A section can’t be allocated, because no configured memory area exists that is large enough to hold it.

Action If you are using a linker command file, check that the MEMORY and SECTIONS directives allow enough room to ensure that no sections are being placed in unconfigured memory.

cannot allocate (...) in (...)

Description A section could not be allocated in the named memory area because the memory area was not large enough to hold the section

Action If you are using a linker command file, check that the MEMORY and SECTIONS directives allow enough room to place the section in the named memory area. Take alignment and blocking restrictions into account.

cannot allocate (...) (page ...)

Description A section could not be allocated because there was no available memory area with the required attribute(s) (R, W, I, or X).

Action If you are using a linker command file, check that there is enough configured memory that possesses the necessary attributes to allow the section to be allocated.

creating output section (...) without SECTIONS directive

Description This message is a warning that the linker is creating an output section that was not specified in the SECTIONS directive. This warning occurs when you use the –w option.

Action Allocate the specified section in the SECTIONS directive.

L

load address for uninitialized section (...) ignored

Description Uninitialized sections have no load addresses—only run addresses.

load address for UNION ignored

Description UNION refers only to the section's run address.

load allocation for GROUP member (...) ignored

Description The entire group is treated as one contiguous unit, so the group itself can be aligned or bound to an address, but the individual sections making up the group cannot be handled individually.

load allocation required for initialized GROUP member (...)

Description GROUP refers to the runtime allocation only. You must specify the load address for all initialized sections within a group separately.

load allocation required for initialized UNION member (...)

Description UNIONs refer to runtime allocation only. You must specify the load address for all sections within a UNION separately.

M

memory types (...) and (...) on page (...) overlap

Description Memory ranges on the same page cannot overlap.

Action If you are using a linker command file, check that MEMORY and SECTIONS directives allow enough room to ensure that no sections are being placed in unconfigured memory.

N

no allocation allowed for uninitialized UNION member

Description An uninitialized section with a UNION gets its run allocation from the UNION and has no load address, so no allocation is valid for the member.

no allocation allowed with a GROUP–allocation for section (...) ignored

Description The entire group is treated as one unit, so the group can be aligned or bound to an address, but the sections making up the group cannot be handled individually.

no load address specified for (...); using run address

Description If an initialized section has a run address only, the section is allocated to run and load at the same address.

no run allocation allowed for union member (...)

Description A UNION defines the run address for all of its members; therefore, individual run allocations are illegal.

O

output file (...) not executable

Description The output file created may have unresolved symbols or other problems stemming from other errors. This condition is not fatal.

P

PC-relative displacement overflow at address (...) in file (...)

Description relocation of a PC-relative jump resulted in a jump displacement too large to encode in the instruction.

PP 15-bit offset overflow at (...) in (...)

Description A scaled offset value designated in a PP addressing mode was greater than a 15-bit value.

Action When linking code generated by the PP compiler, you need to ensure that the .pbss section is allocated into on-chip data RAM. You must also ensure that all of the offsets used in addressing can be scaled to 15 bits. For more information, see subsection 3.1.3, *Static and Global Memory Models.*

R

run allocation for GROUP member (...) ignored

Description The entire group is treated as one contiguous unit, so the group itself can be aligned or bound to an address, but the individual sections making up the group cannot be handled individually.

S

section (...) at (...) overlays at address (...)

Description The two sections overlap and cannot be allocated.

Action If you are using a linker command file, check that the MEMORY and SECTIONS directives allow enough room to ensure that no sections overlap.

section (...) enters unconfigured memory at address (...)

Description A section can't be allocated because no configured memory area exists that is large enough to hold it.

Action If you are using a linker command file, check that the MEMORY and SECTIONS directives allow enough room to ensure that no sections are being placed in unconfigured memory.

section (...) not found

Description An input section specified in a SECTIONS directive was not found in the input file.

section (...) won't fit into configured memory at (...)

Description A section can't be allocated, because no configured memory area exists that is large enough to hold it.

Action If you are using a linker command file, check that MEMORY and SECTIONS directives allow enough room to ensure that no sections are being placed in unconfigured memory.

symbol (...) multiply defined: (...) and (...)

Description The symbol named has been defined in both named files.

U

undefined symbol (...) first referenced in file (...)

Description Unless the –r option is used, the linker requires that all referenced symbols be defined. This condition prevents the creation of an executable output file.

Action Link using the –r option or define the symbol.

15.3 I/O and Internal Overflow Errors

The following error messages indicate that the input file is corrupt, nonexistent, or unreadable, or that the output file cannot be opened or written to. Messages in this category may also indicate that the linker is out of memory or table space.

A

archive symbol ... archive (...)

Description The archive library may be corrupt or improperly built.

Action Try rebuilding the library.

C

cannot create output file (...)

Description This message usually indicates an illegal filename.

Action Check spelling, pathname, environment variables, etc. The filename must conform to operating system conventions.

can’t create map file (...)

Description This message usually indicates an illegal filename.

Action Check spelling, pathname, environment variables, etc. The filename must conform to operating system conventions.

can’t find input file *filename*

Description The file, *filename*, is not in your PATH, is misspelled, etc.

Action Check spelling, pathname, environment variables, etc.

can’t open (...)

Description The specified file does not exist.

Action Check spelling, pathname, environment variables, etc.

can’t read (...)

Description The file may be corrupt.

Action If the input file is corrupt, try reassembling it.

can’t seek (...)

Description The file may be corrupt.

Action If the input file is corrupt, try reassembling it.

can’t write (...)

Description Disk may be full or protected.

Action Check disk volume and protection.

F

fail to read (...)

Description The file may be corrupt.

Action If the input file is corrupt, try reassembling it.

fail to seek (...)

Description The file may be corrupt.

Action If the input file is corrupt, try reassembling it.

fail to write (...)

Description Disk may be full or protected.

Action Check disk volume and protection.

file (...) has incompatible byte ordering

Description You attempted to link two files that have different endian ordering.

Action You must make sure that all files that are linked together have the same byte ordering (endianess).

file (...) has no relocation information

Description You have attempted to relink a file that was not linked with –r.

file (...) is of unknown type

Description The input file may be corrupt

Action Try recompiling or assembling the file.

file (...) is of unknown type, magic number = (...)

Description The binary input file is not a COFF file.

I

illegal relocation type (...) found in section(s) of file (...)

Description The binary file is corrupt.

internal error (...)

Description Indicates the linker has an internal error.

invalid archive size for file (...)

Description The archive file is corrupt.

I/O error on output file (...)

Description Disk may be full or protected.

Action Check disk volume and protection.

L

library (...) member (...) has incompatible byte ordering

Description You attempted to link two files that have different endian ordering.

Action You must make sure that all files that are linked together have the same byte ordering (endianess).

library (...) member (...) has no relocation information

Description The input file may be corrupt

Action Try recompiling or assembling the file.

library (...) member (...) is of unknown type

Description The input file may be corrupt

Action Try recompiling or assembling the file.

line number entry found for absolute symbol

Description The input file may be corrupt.

Action If the input file is corrupt, try reassembling it.

N

no string table in file *filename*

Description The input file may be corrupt.

Action If the input file is corrupt, try reassembling it.

no symbol map produced – not enough memory

Description This is a nonfatal condition that prevents the generation of the symbol list in the map file.

O

out of memory, aborting

Description Your system does not have enough memory to perform all required tasks.

Action Try breaking the assembly language files into multiple smaller files and do partial linking. Refer to Section 13.14, *Partial (Incremental) Linking*.

R

relocation symbol not found: index (...), section (...), file (...)

Description The input file may be corrupt.

Action If the input file is corrupt, try reassembling it.

S

seek to (...) failed

Description The input file may be corrupt.

Action If the input file is corrupt, try reassembling it.

T

too few symbol names in string table for archive (...)

Description The archive file may be corrupt.

Action If the input file is corrupt, try recreating the archive.

Chapter 16

Archiver Description

The TMS320C8x archiver lets you combine several individual files into a single file called an **archive** or a **library**. You can build libraries out of any type of files. Both the assembler and the linker accept archive libraries as input; the assembler can use libraries that contain individual source files, and the linker can use libraries that contain individual object files.

One of the most useful applications of the archiver is building libraries of object modules. For example, you could write several arithmetic routines, assemble them, and then use the archiver to collect the object files into a single, logical group. You can then specify an object library as linker input. The linker will search through the library and include any members that resolve external references.

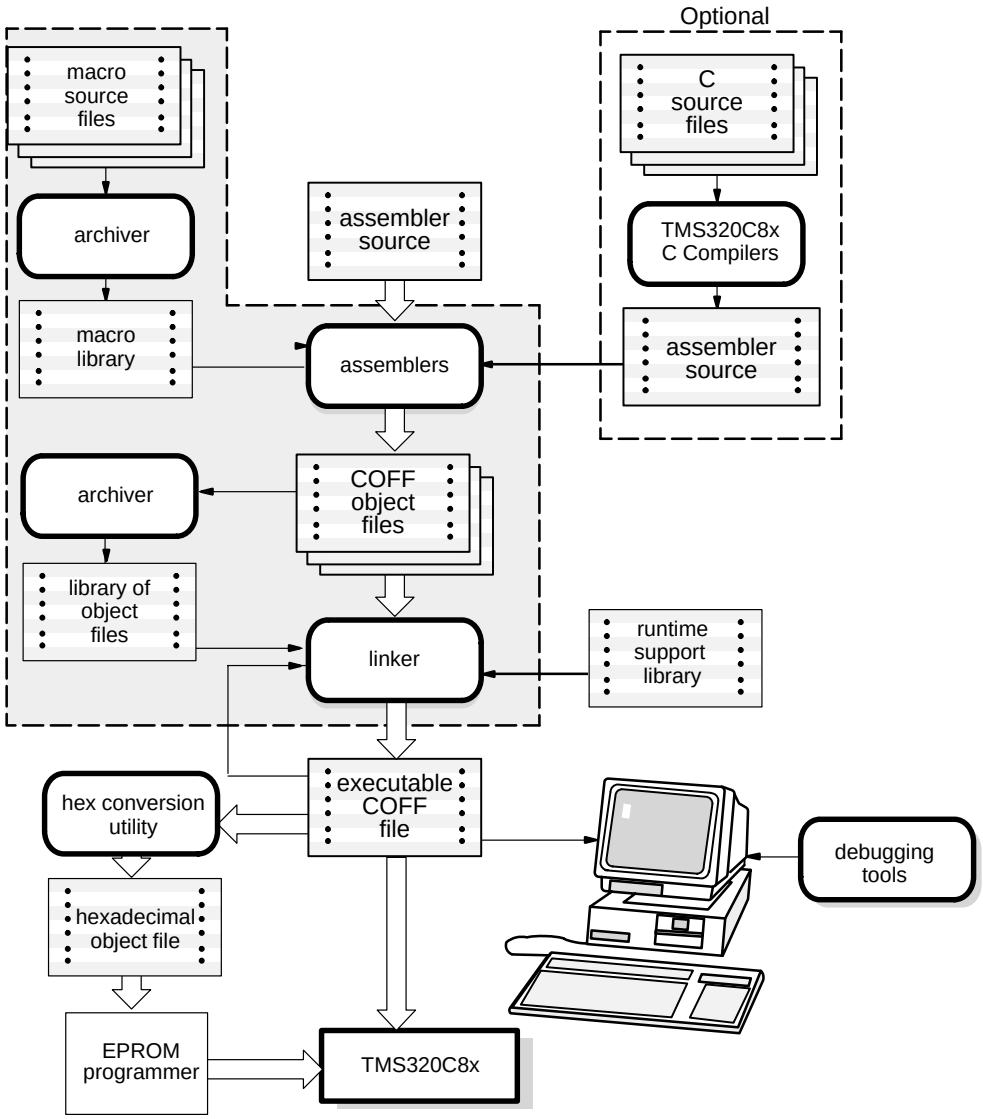
You can also use the archiver to build macro libraries. You can create several separate source files, each of which contains a single macro, and then use the archiver to collect these macros into a single, functional group. The .mlib assembler directive lets you specify the name of a macro library to the assembler; during the assembly process, the assembler will search the specified library for the macros that you call. Chapter 10, *Macro Language*, discusses macro language and macro libraries in detail.

Topics

16.1 Archiver Development Flow

Figure 16–1 shows the archiver's role in the assembly language development process. Both the assembler and the linker accept libraries as input.

Figure 16–1. Archiver Development Flow



16.2 Invoking the Archiver

To invoke the archiver, enter:

mvpar *[–]command[option] libname [filename$_1$ ... filename$_n$]*

mvpar is the command that invokes the archiver.

libname names an archive library. If you don't specify an extension for *libname,* the archiver uses the default extension **.lib**.

filename names individual member files that are associated with the library. If you don't specify an extension for a *filename,* the archiver uses the default extension **.obj**.

command tells the archiver how to manipulate the members in the library. A command can be preceded by an optional hyphen. You **must** use one of the following commands when you invoke the archiver, but you can use only **one** command per invocation. These are valid archiver commands:

- **–a** adds the specified files to the library. Note that this command **does not replace** an existing member that has the same name as an added file; it simply *appends* new members to the end of the archive. It is possible to have several members with the same name in an archive. If you want to *replace* existing members, use the **r** command.
- **–d** deletes the specified members from the library.
- **–r** replaces the specified members in the library. If you don't specify any filenames, the archiver replaces the library members with files of the same name in the current directory. If the specified file is not found in the library, the archiver adds it instead of replacing it.
- **–t** prints a table of contents of the library. If you specify filenames, only those files are listed. If you don't specify any filenames, the archiver lists all the members in the specified library.

–x extracts the specified files. If you don't specify any member names, the archiver extracts all the members in the library. When the archiver extracts a member, it simply copies the member into the current directory; it *doesn't* remove it from the library.

In addition to the *commands,* you can specify the following *options*:

–e tells the archiver not to use the default extension .obj for member names.

–q (quiet) suppresses the banner and status messages.

–s prints a list of the global symbols that are defined in the library. (This option is valid only with the –a, –r, and –d commands.)

–v (verbose) provides a file-by-file description of the modifications to a library and its constituent members.

Note: Naming Library Members

It is possible (but not desirable) for a library to contain several members with the same name. If you attempt to delete, replace, or extract a member, and the library contains more than one member with the specified name, then the archiver deletes, replaces, or extracts the first member with that name.

16.3 Archiver Examples

Here are some examples of archiver use:

❑ Example 1

This example creates a library called function.lib that contains the files sine.obj, cos.obj, and flt.obj.

```
mvpar –a function sine cos flt

MVP archiver                Version x.xx
Copyright (c) 1993–1995     Texas Instruments Incorporated
  ==>  new archive 'function.lib'
  ==>  building archive 'function.lib'
```

Because these examples use the default extensions (.lib for the library and .obj for the members), it is not necessary to specify them.

❑ Example 2

You can print a table of contents of function.lib with the –t option:

```
mvpar –t function

MVP Archiver                Version x.xx
Copyright (c) 1993–1995     Texas Instruments Incorporated

          FILE NAME     SIZE   DATE
  ----------------    -----   ------------------------
           sine.obj       310   Wed Jul 27 12:14:39 1994
            cos.obj       294   Wed Jul 27 13:28:21 1994
            flt.obj       428   Wed Jul 27 09:59:22 1994
```

❑ Example 3

You can explicitly specify extensions if you don't want the archiver to use the default extensions; for example:

```
mvpar –av function.fn sine.asm cos.asm flt.asm
MVP Archiver                Version x.xx
Copyright (c) 1993–1995     Texas Instruments Incorporated
   ==>   add 'sine.asm'
   ==>   add 'cos.asm'
   ==>   add 'flt.asm'
   ==>   building archive 'function.fn'
```

This creates a library called function.fn that contains the files sine.asm, cos.asm, and flt.asm. (–v is the verbose option.)

❑ Example 4

If you want to add new members to the library, specify:

```
mvpar –as function tan.obj arctan.obj area.obj
```

```
MVP Archiver                Version x.xx
Copyright (c) 1993–1995     Texas Instruments Incorporated
   ==>   symbol defined: 'K2'
   ==>   symbol defined: 'Rossignol'
   ==>   building archive 'function.lib'
```

Because this example doesn't specify an extension for the libname, the archiver adds the files to the library called function.lib. If function.lib didn't exist, the archiver would create it. (The –s option tells the archiver to list the global symbols that are defined in the library.)

❑ Example 5

If you want to modify a member in a library, you can extract it, edit it, and replace it. In this example, assume there's a library named macros.lib that contains member push.asm.

```
mvpar –x macros push.asm
```

The archiver makes a copy of push.asm and places it in the current directory; it doesn't remove push.asm from the library, though. Now you can edit the extracted file. To replace the copy of push.asm that's in the library with the edited copy, enter:

```
mvpar –r macros push.asm
```

Chapter 17

Hex Conversion Utility Description

The TMS320C8x assemblers and linker create object files that are in common object file format (COFF). Most EPROM programmers do not accept COFF object files as input. The hex conversion utility translates a COFF object file into one of several standard ASCII hexadecimal formats suitable for loading into an EPROM programmer. The utility is also useful in other applications requiring a hexadecimal translation of a COFF object file (for example, when using debuggers and loaders).

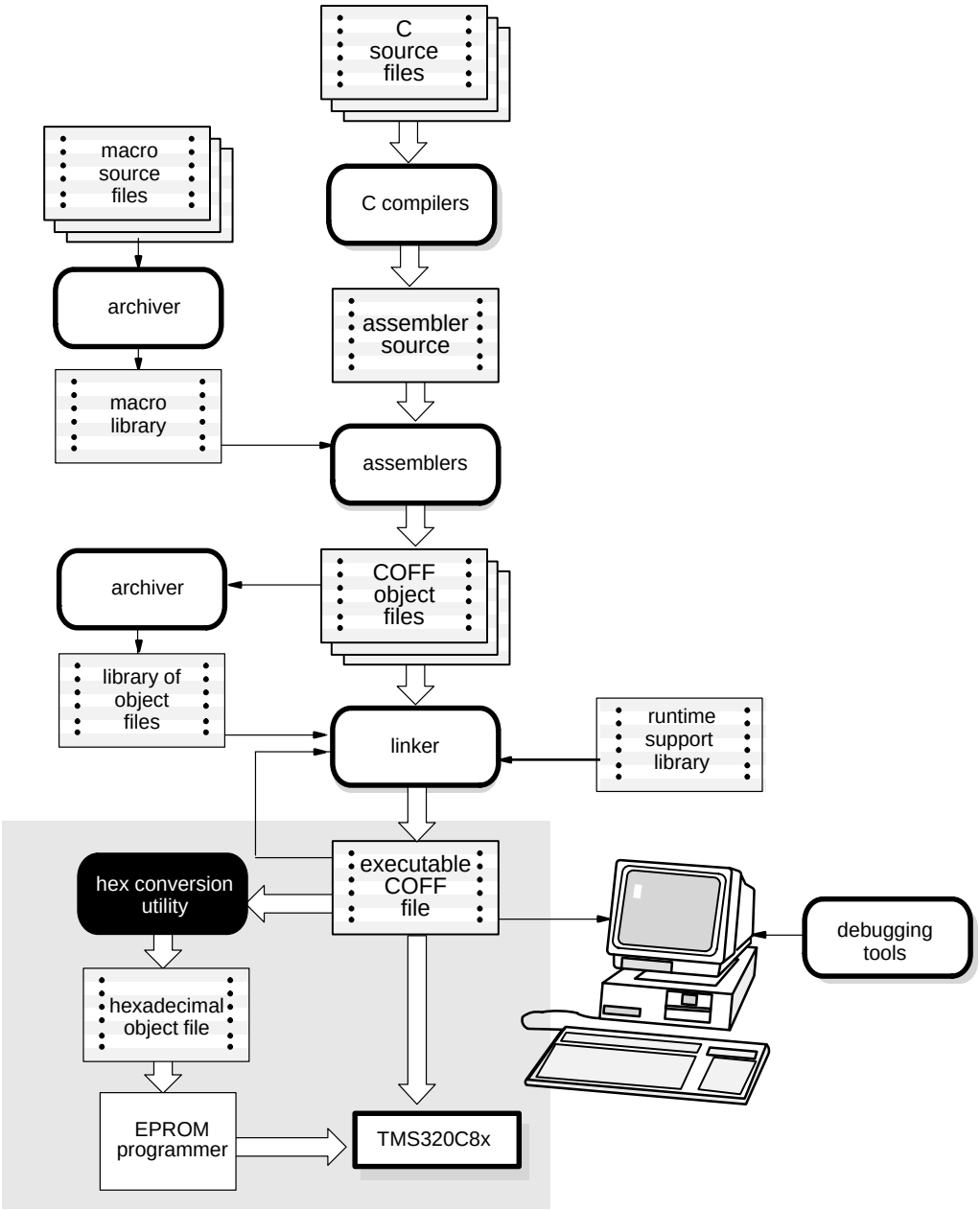
This chapter describes the operation of the hex conversion utility.

Topics

17.1 Hex Conversion Utility Development Flow

Figure 17–1 illustrates the role of the hex conversion utility in the assembly language development process.

Figure 17–1. Hex Conversion Utility Development Flow



The hex conversion utility can produce output files in the following formats:

- ASCII-Hex, supporting 16-bit addresses
- Extended Tektronix (Tektronix), supporting 32-bit addresses
- Intel MCS-86 (Intel), supporting 32-bit addresses
- Motorola Exorciser (Motorola-S), supporting 16-bit addresses
- Texas Instruments SDSMAC (TI-Tagged), supporting 16-bit addresses

17.2 Invoking the Hex Conversion Utility

To invoke the hex conversion utility, enter the following:

mvphex [*–options*] *filename*

mvphex is the command that invokes the hex conversion utility.

–options supply the utility with additional information (see Table 17–1 and the subsections following).

Options control the hex conversion process. You can use options on the command line or in a command file.

- All options are preceded by a dash and are not case-sensitive.
- Several options have an additional parameter that must by separated from the option by at least one space.
- Options with multicharacter names must be spelled exactly as they are shown; no abbreviations are allowed.
- Options are not affected by the order in which they are used. The exception to this rule is the –q option, which must be used before any other options.

filename names a COFF object file or a command file (for more information on command files, see Section 17.4, page CG:17-14).

Table 17–1. Options

General Options	Option	Description	Page
These options control the overall operation of the hex conversion utility.	–map *filename*	Generate a map file	CG:17-20
	–o *filename*	Specify an output filename	CG:17-24
	–q	Generate a quiet run (when used, it must appear *before* other options)	CG:17-14

Table 17–1. Options (Continued)

Image Options	Option	Description	Page
These options allow you to create a continuous image of a range of target memory.	–byte	Number bytes sequentially in image mode	CG:17-30
	–fill *value*	Fill holes in image mode with a value (default value = 0)	CG:17-27
	–image	Specify image mode	CG:17-26
	–zero	Reset the address origin to zero in image mode	CG:17-29
Memory Options	**Option**	**Description**	**Page**
These options allow you to configure the memory widths for your output files.	–datawidth *value*	Define the logical data word width (default = 64 bits)	CG:17-7
	–memwidth *value*	Define the system memory word width (default = 32 bits)	CG:17-8
	–order {LS | MS}	Specify the memory word ordering	CG:17-12
	–romwidth *value*	Specify the ROM device width in bits (default depends on format used)	CG:17-10
Format Options	**Option**	**Description**	**Page**
These options allow you to specify the output format.	–a	Select ASCII-Hex format	CG:17-33
	–i	Select Intel format	CG:17-34
	–m	Select Motorola-S format	CG:17-35
	–t	Select TI-Tagged format	CG:17-36
	–x	Select Tektronix format	CG:17-37

There are two basic methods for invoking the utility:

- **Specify the options and filenames on the command line.** You can enter the command and associated options on the command line to invoke the utility. The following example converts the file firmware.out into TI-Tagged format, producing two output files: firm.lsb and firm.msb.

  ```
  mvphex –t firmware –o firm.lsb –o firm.msb
  ```

- **Specify the options and filenames in a command file.** You can create a batch file that contains the options and filenames you want to use with the hex conversion utility. The following example invokes the utility using a command file called hexutil.cmd :

  ```
  mvphex hexutil.cmd
  ```

  For a full discussion about command files, see Section 17.4, page CG:17-14.

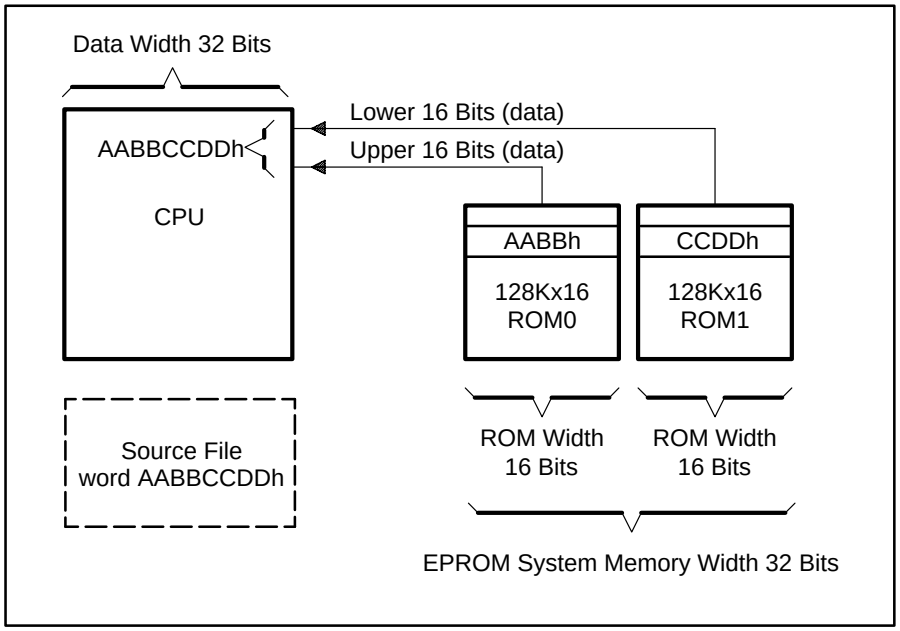
17.3 Understanding Memory Widths

The hex conversion utility provides full flexibility for different memory architectures. In order to use the hex conversion utility, **you must understand how the utility treats word widths**. A word width is the number of bits in a word. Four widths are important in the conversion process: target width, data width, memory width, and ROM width.

When we refer to target word, data word, memory word and ROM word, we refer to a word of such a width.

Figure 17–2 shows a typical memory configuration example. The memory system consist of two 128K x 16 ROM devices. The hex conversion utility provides full flexibility for different memory architectures.

Figure 17–2. Memory Configuration Example



Target Width

Target width is the unit size (in bits) of raw data fields in the COFF file. This generally corresponds to the size of an opcode on the target processor. The width is fixed for each target and cannot be changed. The TMS320C8x has a 64-bit width.

Data Width

Data width is the logical width, in bits, of the data words stored in a particular section of the COFF file. Usually, the logical data width is the same as the width of the target processor. However, since the TMS320C8x processors are byte addressable, the width of a data word can be narrower than the width of the processor.

Note: Data Width Should Equal Target Width

For the TMS320C8x, the data width should always be equal to the target width, which is 64-bits. The data width option is provided for compatibility with other architectures.

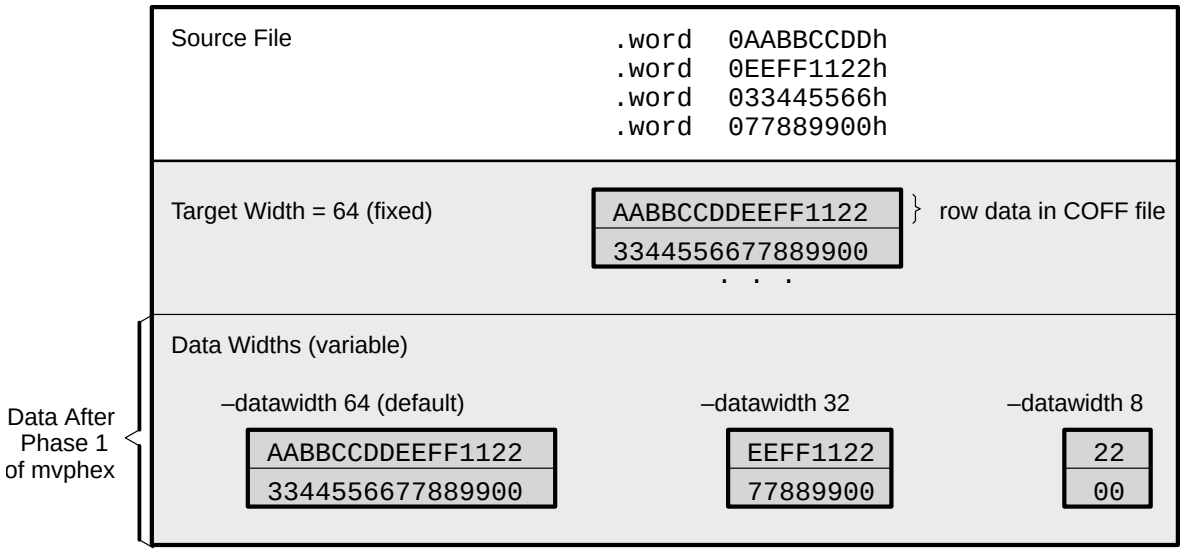
The data width option allows the hex conversion utility to extract the appropriate number of bits from the COFF file for each data word (for example, 16), and only store those bits, even though the COFF file uses 64 bits to store each data word. The hex conversion utility simply truncates the COFF word to the specified length, preserving only the least significant bits of the COFF word.

You can change the data width by:

- Using the **–datawidth** *value* option. This changes the size of data words inside all sections in a COFF file. For example, –datawidth 8 changes the size of the data words to 8 bits.
- Setting the **datawidth=***value* parameter inside the SECTIONS directive. This changes the size of data words for the specific section and overrides the **–datawidth** option for that section. Refer to Section 17.6 for details.

Figure 17–3 shows how the data width is related to the target width.

Figure 17–3. Target and Data Widths



Memory Width

Memory width is the physical width, in bits, of the memory system. Usually, the memory system is physically the same width as the target processor width: a 64-bit processor has a 64-bit memory architecture.

The hex utility defaults memory width to the target width (in this case, 64 bits).

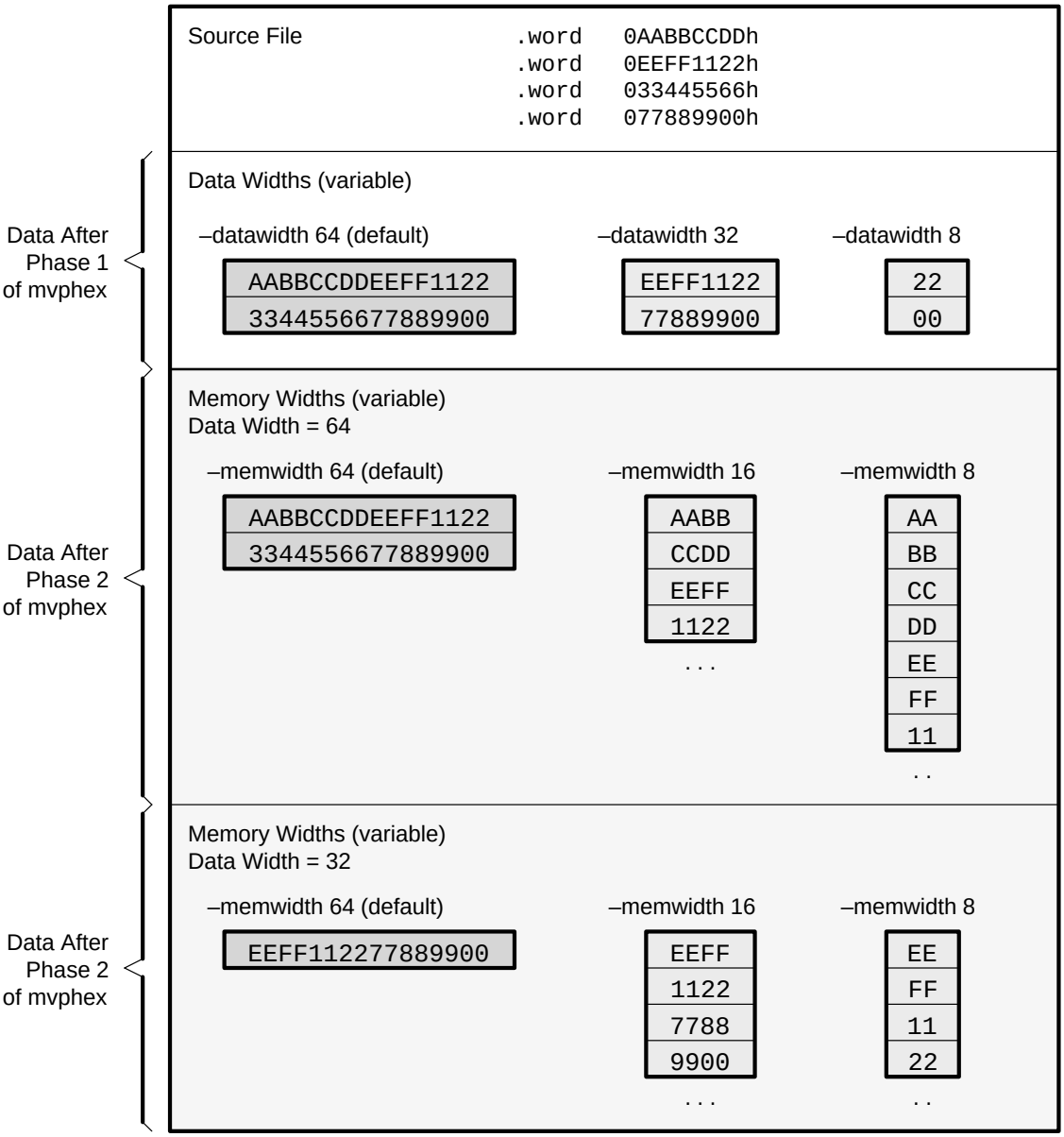
The –memwidth option defines the physical width, in bits, of the memory system. You can change the memory width by:

- Using the **–memwidth** *value* option. This changes the memory width value for the entire file, or
- Setting the **memwidth=***value* parameter inside the ROMS directive. This changes the memory width for the address range specified in the ROMS directive and overrides the –memwidth option for that range. See Section 17.5, page CG:17-16.

The value used should be a multiple of eight bits.

Figure 17–4 demonstrates how the memory width is related to the target width.

Figure 17–4. Target, Data, and Memory Widths



The ROM Width

The ROM width specifies the physical width, in bits of each ROM device and corresponding output file, (usually one byte or eight bits).

The ROM width determines how the hex utility partitions the data into output files. After the target words are mapped to the memory words, the memory words are broken into one or more output files. The number of output files is determined by the following formula:

number of files = memory width ÷ ROM width

For example, for the ’C8x, you could specify –romwidth 64 and get a single output file containing 64-bit words. Or you can use –romwidth 16 to get four files, each containing 16 bits of each word.

The default ROM width value that the hex utility uses depends on the output format:

- All hex formats except TI-Tagged are oriented as lists of 8-bit bytes; the default ROM width for these formats is 8.
- TI-Tagged is a 16-bit format; the default ROM width for TI-Tagged is 16.

You can change the ROM width by:

- Using the **–romwidth** *value* option. This changes the ROM width value for the entire COFF file, or
- Setting the **romwidth=***value* parameter inside the ROMS directive. This changes the ROM width value for a specific ROM address range and overrides the –romwidth option for that range. See Section 17.5, page CG:17-16.

The value used should be multiple of eight bits.

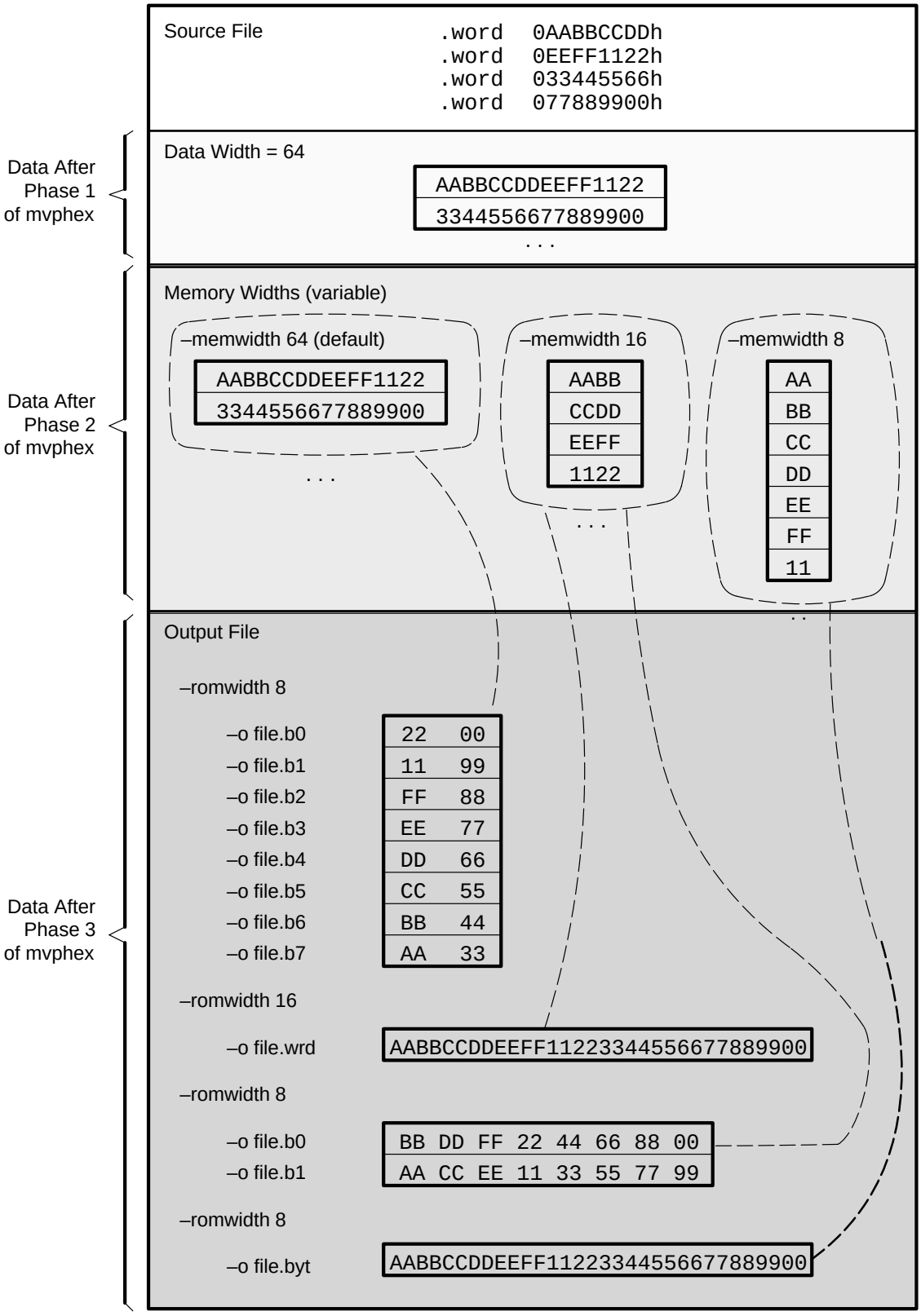
Note: The TI-Tagged Format is 16 Bits Wide

You cannot use the –romwidth option to change the width of the TI-Tagged format. The TI-Tagged format supports a 16-bit ROM width only.

If you select a ROM width that is wider than the natural size of the output format (16 for TI-Tagged or 8 for all other hex formats), the utility simply writes multibyte fields into the file.

Figure 17–5 illustrates how the target, memory, and ROM widths are related to one another.

Figure 17–5. Target, Memory, and ROM Widths



Ordering Memory Words

When memory words are narrower than target words, target words are split into multiple consecutive memory words. There are two ways to split a wide word into consecutive memory locations in the same hex utility output file:

- By using **big-endian** ordering, in which the most significant part of the wide word occupies the first of the consecutive locations
- By using **little-endian** ordering, in which the the least significant part occupies the first location

The –order option

You can use the –order option to specify the ordering of the memory words: **–order** *MS* for big-endian (default) and **–order** *LS* for little-endian.

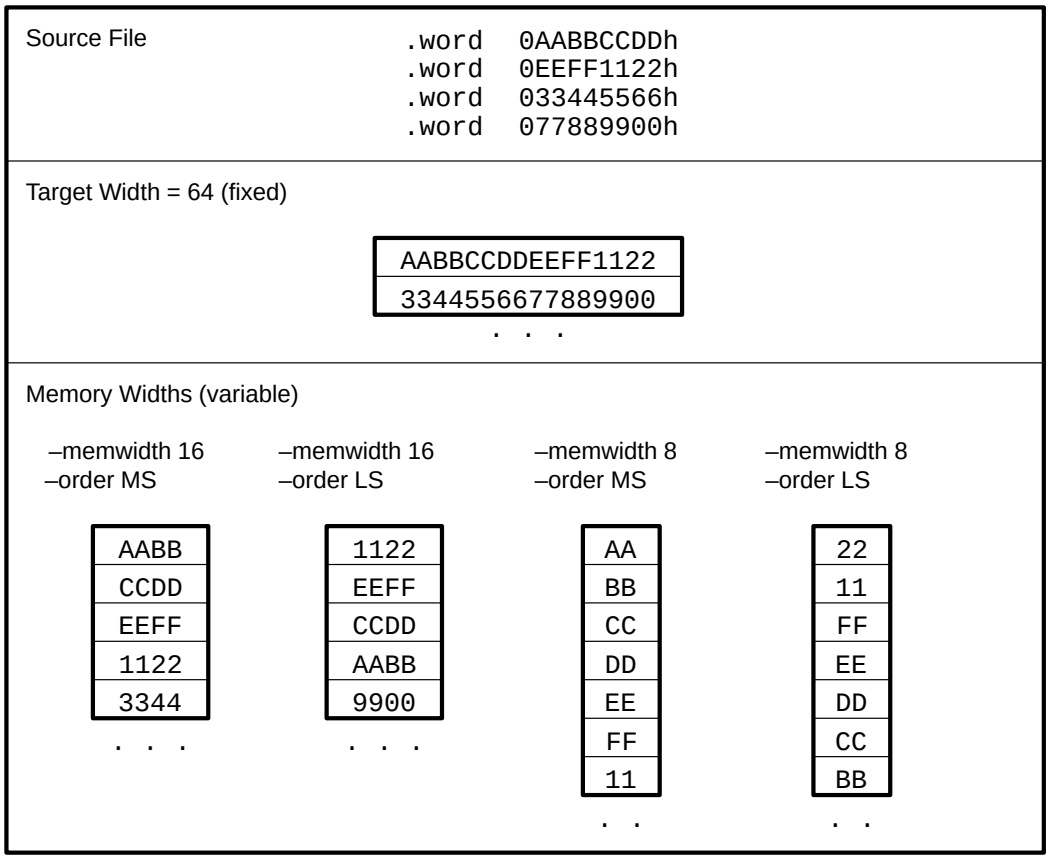
Unless you use the –order option, the utility uses big-endian format for the default ordering.

Note: When the –order Option Applies

- The –order option applies only when you use the –memwidth option with a value less than 64. Otherwise, the –order is ignored.
- The –order option does not affect how memory words are split into output files. Think of the files as a set: the set contains a least significant file and a most significant file, but there is no ordering over the set. When you list filenames for a set of files, you *always* list the least significant first, regardless of the –order option.

Figure 17–6 demonstrates how –order LS and –order MS affect the conversion process. This figure, and the previous figure, Figure 17–5, explain the condition of the data in the hex utility output files.

Figure 17–6. Varying The Word Order



17.4 Using Command Files

The hex conversion utility allows you to store conversion parameters in a command file. This is useful if you plan to invoke the utility more than once with the same input files and options. A command file also allows you to use the ROMS and SECTIONS directives to customize the conversion process. (For more information about the ROMS or SECTIONS directives, see Section 17.5 or 17.6, respectively.)

Command files are ASCII files that contain one or more of the following:

- **Options.** These are specified in a command file in exactly the same manner as on the command line. For a complete listing of general utility options, see Table 17–1, page CG:17-4, and Table 17–2, page CG:17-32.

- **ROMS and SECTIONS directives.** The ROMS directive allows you to specify the physical memory configuration of your system as a list of address-range parameters. The SECTIONS directive allows you to specify which sections from the COFF object file should be selected.

- **Comments.** You can add comments to your command file by using the /* and */ delimiters.

To invoke the utility and use the options you defined in a command file, enter the command filename on the command line:

mvphex *commandfilename*

You can also specify other options and files on the command line. For example, you could invoke the utility using both a command file and command line options:

```
mvphex firmware.cmd –map firmware.mxp
```

The order in which these options and file names appear is not important. The utility reads all input from the command line and all information from the command file before starting the conversion process. However, if you are using the –q option, *it must appear as the first option on the command line or in a command file.*

The –q option suppresses the utility's normal banner and progress information, which includes the name of the output format and a line showing each section as it is read and converted.

Examples of Command Files

❑ Example 1

The file firmware.cmd contains the following lines:

```
firmware.out /* input file   */
–t           /* TI–Tagged    */
–o firm.b0   /* output file  */
–o firm.b1   /* output file  */
–o firm.b2   /* output file  */
–o firm.b3   /* output file  */
```

You can invoke the hex conversion utility using the command above file with the following command:

```
mvphex firmware.cmd
```

Invoking the utility and using this command file is the same as entering the following on the command line:

```
mvphex –t firmware –o firm.b0 –o firm.b1 –o firm.b2 –o firm.b3
```

❑ Example 2

The example below converts a file called appl.out into eight hex files in Intel format. Each output file is two bytes wide and 8K bytes long.

```
/************************************************/
/* HEX CONVERSION UTILITY – COMMAND FILE EXAMPLE*/
/************************************************/
appl.out         /* input file    */
–i               /* Intel format  */
–map appl.mxp    /* map file      */

ROMS
{
   ROW1: origin=02000000h len=04000h romwidth=16
         files={ appl.u0 appl.u1 appl.u2 appl.u3 }
   ROW2: origin 02004000h len=04000h romwidth=16
         files={ app1.u4 appl.u5 appl.u6 appl.u7 }
}

SECTIONS
{
   .text, .data, .cinit, .sect1, .vectors, .const:
}
```

17.5 The ROMS Directive

The ROMS directive allows you to specify the physical memory configuration of your system as a list of address-range parameters.

The ROMS directive also allows you to partition the hex output into different files on address boundaries. Each file will then be used to program one single ROM device.

The ROMS directive is syntactically similar to the MEMORY directive of the TMS320C8x linker: both define the memory map in the target address space. Each line entry in the ROMS directive defines a specific address range. The general syntax is:

```
ROMS
{
        name:[origin=value,][length=value,]
        [romwidth=value,][memwidth=value,]
        [fill=value,] [files={filename1, filename2, ...}]
...
}
```

ROMS is the keyword identifying the directive.

name identifies a memory range. The name of the memory range may be one to eight characters in length. The name has no significance to the program; it simply identifies the range. Duplicate memory range names are allowed.

origin specifies the starting address of a memory range. It may be entered as origin, org, or o. The associated value must be a decimal, octal, or hex constant. If you omit the origin value, it defaults to 0.

The following table summarizes the notation you can use to specify a decimal, octal, or hex constant:

Constant	Notation	Example
Hexadecimal	0x prefix or h suffix	0x77 or 077h
Octal	0 prefix	077
Decimal	No prefix or suffix	77

length specifies the length of a memory range in units equal to the width of the ROM. It may be entered as length, len, or l. The value must be a decimal, octal, or hex constant. If you omit the length value, it defaults to the rest of the address space.

romwidth specifies the physical ROM width of this range in bits (see page CG:17-10). Any value you specify here overrides the –romwidth option. The value must be a decimal, octal, or hex constant, and be a multiple of eight.

memwidth specifies the memory width of this range in bits (see page CG:17-8). Any value you specify here overrides the –memwidth option. The value must be a decimal, octal, or hex constant and be a multiple of eight. *When using this option, you need to specify the paddr parameter for each section in the SECTIONS directive.*

fill specifies a fill value to use for this range. In image mode, the hex conversion utility fills any holes between sections in a range with this value. The value should be a decimal, octal, or hex constant with a width equal to the target width. Any value you specify here overrides the –fill option. You need to use the –image option when using –fill. See page CG:17-26.

files identify the names of the output files that correspond to this range. Enclose the list of names in curly braces and order them from *least significant* to *most significant* output file.

The number of file names should equal the number of output files the range will generate. To calculate the number of output files, refer to Section 17.3. The utility warns you if you list too many or too few filenames.

Unless you are using image mode (–image option), all of the parameters defining a range are optional. A range with no origin or length defines the whole address space. In image mode, an origin and length are required for all ranges. The commas and equals signs are also optional.

When To Use The ROMS Directive

If you don't use a ROMS directive, the utility defines a single default range that includes the entire program address space (PAGE 0). This is equivalent to a ROMS directive with a single range without origin or length.

Use the ROMS Directive when you want to:

- **Program data into fixed-size ROMs**. You can use the ROMS directive to program large amounts of data into fixed-size ROMs. When you specify memory ranges corresponding to the length of your ROMs, the utility automatically breaks the output into blocks that fit into the ROMs.
- **Restrict output to certain segments**. You can also use the ROMS directive to restrict the conversion to a certain segment or segments of the target address space. The utility does not convert the data that falls outside of the ranges defined by the ROMS directive. Sections can span range boundaries; the utility splits them at the boundary into multiple ranges. If a section falls completely outside any of the ranges you define, the utility does not convert that section and issues no messages or warnings. In this way, you can exclude sections without listing them by name with the SECTIONS directive. However, if a section falls partially in a range and partially in unconfigured memory, the utility issues a warning and converts only the part within the range.
- **Use image mode.** If you use the –image option, you must use a ROMS directive. Each range is filled completely so that each output file in a range contains data for the whole range. Gaps before, between or after sections are filled with the fill value from the ROMS directive, with the value specified with the –fill option, or with the default value of zero.

An Example of the ROMS Directive

The ROMS directive in Figure 17–7 shows how 16K words of 32-bit memory could be partitioned for eight 8K x 8 EPROMs.

Figure 17–7. A ROMS Directive Example

```
/*************************************************/
/* Sample command file with ROMS directive       */
/*************************************************/
infile.out
–image
–memwidth 32

ROMS
{
   EPROM1: org = 04000h, len = 02000h, romwidth = 8
           files = { rom4000.b0, rom4000.b1,
                     rom4000.b2, rom4000.b3 }

   EPROM2: org = 06000h, len = 02000h, romwidth = 8,
           fill = 0FFh,
           files = { rom6000.b0, rom6000.b1,
                     rom6000.b2, rom6000.b3 }
}
```

EPROM1 defines the address range from 4000h through 5FFFFh. The range contains the following sections:

This section	Has this range
.text	4000h through 487Fh
.data	5B80H through 5FFFh

The rest of the range is filled with 0h (the default fill value). The data from this range is converted into four output files:

- rom4000.b0 contains bits 0 through 7
- rom4000.b1 contains bits 8 through 15
- rom4000.b2 contains bits 16 through 23
- rom4000.b3 contains bits 24 through 31

EPROM2 defines the address range from 6000h through 7FFFh. The range contains the following section:

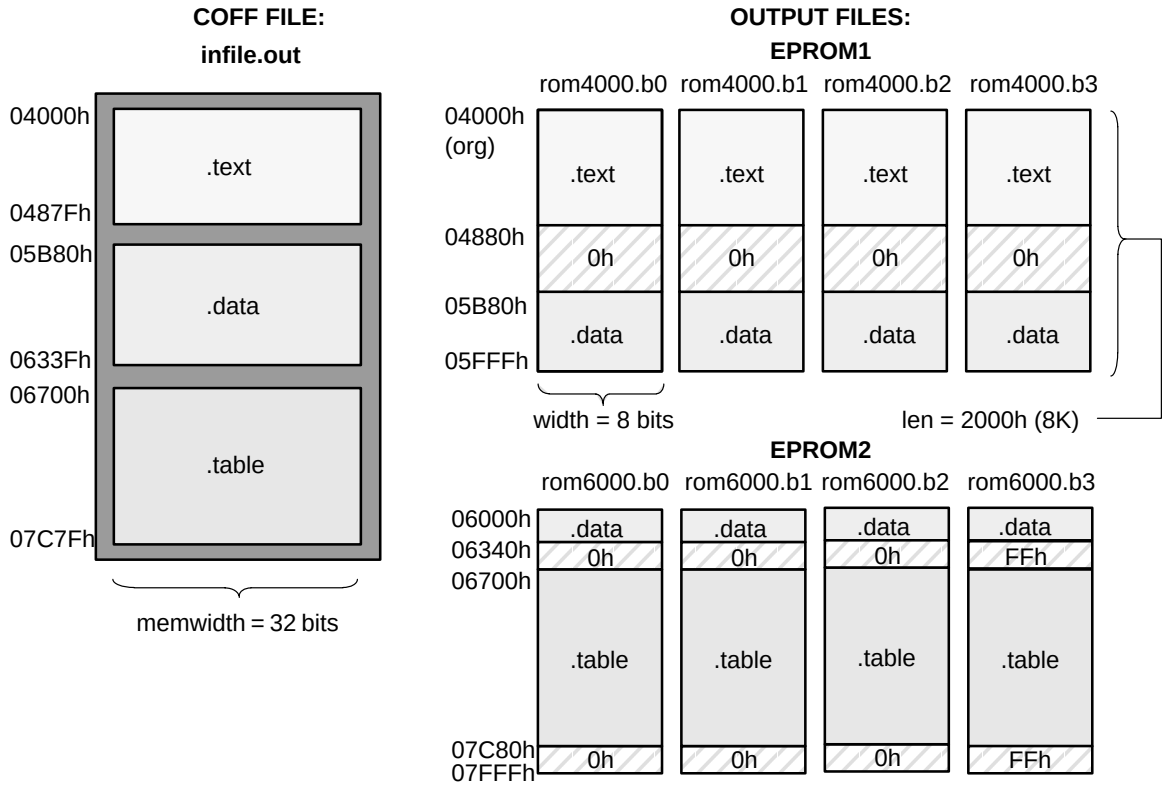
This section	Has this range
.data	6000h through 633Fh
.table	6700h through 7C7Fh

The rest of the range is filled with OFFh (from the specified fill value.) The data from this range is converted into four output files:

- rom6000.b0 contains bits 0 through 7
- rom6000.b1 contains bits 8 through 15
- rom6000.b2 contains bits 16 through 23
- rom6000.b0 contains bits 24 through 31

Figure 17–8 shows how the ROMS directive partitions the infile.out file into eight output files.

Figure 17–8. The infile.out File Partitioned Into Eight Output Files



The –map option

The map file (specified with the –map option) is very useful when you use the ROMS directive with multiple ranges. The map file shows each range, its parameters, names of its associated output files, and a list of contents (section names and fill values) broken down by address. Here's a segment of the map file resulting from the example in Figure 17–7.

Figure 17–9. Map File Output Showing Memory Ranges

```
-----------------------------------------------------
00004000..00005fff  Page=0  Width=8  ”EPROM1”
-----------------------------------------------------
   OUTPUT FILES:   rom4000.b0   [b0..b7]
                   rom4000.b1   [b8..b15]
                   rom4000.b2   [b16..b23]
                   rom4000.b3   [b24..b31]

   CONTENTS: 00004000..0000487f .text
             00004880..00005b7f FILL = 00000000
             00005b80..00005fff .data
-----------------------------------------------------
00006000..00007fff  Page=0  Width=8  ”EPROM2”
-----------------------------------------------------
   OUTPUT FILES:   rom6000.b0   [b0..b7]
                   rom6000.b1   [b8..b15]
                   rom6000.b2   [b16..b23]
                   rom6000.b3   [b24..b31]

   CONTENTS: 00006000..0000633f .data
             00006340..000066ff FILL = 000000ff
             00006700..00007c7f .table
             00007c80..00007fff FILL = 000000ff
```

17.6 The SECTIONS Directive

You may convert specific sections of the COFF file by name with the SECTIONS directive. You can also specify those sections which you wish to locate in ROM at a different address than the *load* address specified in the linker command file.

The concept of the SECTIONS directive is simple:

- If you use a SECTIONS directive, the utility converts only the sections that you list in the directive; the utility ignores any other sections in the COFF file that you don't specify in the directive.
- If you don't use a SECTIONS directive, the utility converts any initialized section that falls within the configured memory. Standard TMS320C8x C-compiler generated initialized sections include: .text, .ptext, .const, .cinit, .pcinit, etc.

Note: Converting Uninitialized Sections

Uninitialized sections are *never* converted, whether or not you specify them in a SECTIONS directive. Standard C-Compiler uninitialized sections include: .bss, .pbss, .stack, .pstack, .sysmem and .psysmem.

You use the SECTIONS directive in a command file. (For more information about using a command file, see Section 17.4, page CG:17-14.) The general syntax for the SECTIONS directive is:

```
SECTIONS
{
        sname: [paddr=value] [datawidth=value],
        sname2: [paddr=value] [datawidth=value],
        ...
}
```

SECTIONS	is the keyword identifying the directive.
sname	identifies a section in the COFF input file. If you specify a section that doesn't exist, the utility issues a warning and ignores the name.

datawidth=*value* specifies the logical width of the data in the section, see page CG:17-7.

paddr=*value* specifies the physical ROM address at which this section should be located. This value overrides the section load address given by the linker. (Refer to Section 17.9).The value here must be a decimal, octal, or hex constant. If one section uses a –paddr option, then all sections must use a –paddr option.

The commas separating section names are optional. For more similarity with the linker's SECTIONS directive, you can use colons after the section names.

In the following example, the COFF file contains six initialized sections: .text, .data, .const, .vectors, .coeff, and .tables. Suppose you only want .text and .data to be converted. Use a SECTIONS directive to specify the sections to be converted:

```
SECTIONS { .text, .data }
```

17.7 Output Filenames

When the hex conversion utility translates your COFF object file into a data format, it partitions the data. This partitioning may cause the utility to produce more than one output file. When multiple files are formed by splitting data into byte-wide or word-wide files, *filenames are always assigned in order from least to most significant.* This is true regardless of target or COFF endian ordering, or of any –order option.

Assigning Output Filenames

The hex conversion utility follows the sequence below to assign output filenames:

1) **Look for the ROMS directive.** If the file associated with a range in the ROMS directive and you have included a list of files (files = {. . .}) on that range, the utility takes the filename from the list.

   In this example, assume that the target data is 64-bit words being converted to four files, 16 bits wide. To name the output files using the ROMS directive, you could use the following:

```
ROMS
{
 RANGE1: romwidth=16, files={ xyz.b0 xyz.b1 xyz.b2 xyz.b3}
}
```

   The utility creates the output files by writing the least significant bits (LSBs) to xyz.b0 and the most significant bits (MSBs) to xyz.b3.

2) **Look for the –o options.** The hex conversion utility often splits the target data from the COFF object file on byte, word, or half-word boundaries. It also partitions the data into files of fixed length according to the configuration you specify. This results in multiple output files. **You can specify names for the output files by using the –o option.** If there are no filenames listed in the ROMS directive and you used –o options, the utility takes the filename from the list of –o options. You can use the –o options either on the command line or in a command file. The following line has the same effect as the example above using the ROMS directive:

```
–o xyz.b0 –o xyz.b1 –o xyz.b2 –o xyz.b3
```

   Note that if both the ROMS directive and –o options are used together, the ROMS directive overrides the –o options.

3) **Assign a default filename.** If the list of –o options is exhausted or if you have not specified output filenames in the ROMS directive or with –o options, the utility assigns a default filename.

Default Filenames

A default filename consists of the base name from the COFF input file plus a two- to three-character extension. The extension characters consist of the following three elements:

1) A format character, based on the output format:

   a for ASCII-Hex
   i for Intel
   t for TI-Tagged
   m for Motorola-S
   x for Tektronix

2) The range number in the ROMS directive. Ranges are numbered starting with 0. If there is no ROMS directive, or only one range, the utility omits this character.

3) The file number in the set of files for the range, starting with 0 for the least significant file.

For example, assume coff.out is for a 32-bit target processor. With no command file (and no ROMS directive), the following command produces four output files named coff.i0, coff.i1, coff.i2, and coff.i3:

```
mvphex –i coff.out
```

If you include the following ROMS directive when you invoke the hex conversion utility, you would have eight output files:

```
ROMS
{
   range1: o = 1000h 1 = 1000h
   range2: o = 2000h 1 = 1000h
}
```

These Files	Contain this data
coff.i00, coff.i01, coff.i02, and coff.i03	1000h through 1FFFh
coff.i10, coff.i11, coff.i12, and coff.i13	2000h through 2FFFh

17.8 Image Mode and the –fill Command

This section explains the advantages of operating in image mode, and details the method of producing output files with a precise, continuous image of a target memory range.

The –image option

With the –image option, the utility generates a memory image by completely filling all of the mapped ranges specified in the ROMS directive. By using the –image option, you can specify image mode. In image mode, the hex conversion utility produces output files that each contain an exact, continuous image of some range of target memory. This is useful when you are using the output files to program ROM devices. If you decide to use image mode, you must create a ROMS directive in a command file to specify the ranges of target memory (see Section 17.5, page CG:17-16, for a description of the ROMS directive).

A COFF file consists of blocks of memory (sections) with assigned memory locations. Typically, all sections are not adjacent: there are gaps between sections in the address space for which there is no data. When such a file is converted *without* the use of image mode, the hex conversion utility bridges these gaps by using the address records in the output file to skip ahead to the start of the next section. In other words, there may be discontinuities in the output file addresses. Discontinuities can be inconvenient when you are trying to program a ROM.

In image mode, there are no discontinuities. Each output file contains a continuous stream of data that corresponds exactly to an address range in target memory. Any gaps before, between, or after sections are filled with a fill value that you supply.

An output file converted by using image mode still has address records because many of the hex formats require an address on each line. However, in image mode, these addresses will always be contiguous.

Note: Defining the Ranges of Target Memory

If you use image mode, you must also use a ROMS directive. In image mode, each output file corresponds directly to a range of target memory. You must define the ranges. If you don't supply the ranges of target memory, the utility tries to build a memory image of the entire target processor address space—potentially a huge amount of output data. To prevent this situation, the utility requires you to explicitly restrict the address space with the ROMS directive.

The –fill option

The –fill option specifies a value for filling the holes between sections in image mode. The fill value must be specified as an integer constant following the –fill option. The width of the constant is assumed to be that of a word on the target processor. For example, specifying –fill 0FFFFh results in a fill pattern of 000000000000FFFFh. The constant value is not sign-extended.

The hex conversion utility uses a default fill value of zero if you don't specify a value with the fill option. *The –fill option is valid only in image mode*; in normal mode, it is ignored.

To use image mode, follow the steps below:

Step 1: Define the ranges of target memory with a ROMS directive (see Section 17.5).

Step 2: Invoke the hex conversion utility with the –image option. You can optionally use the –byte option to number the bytes sequentially and the –zero option to reset the address origin to zero for each output file. If you didn't specify a fill value in the ROMS directive and you want a value other than the default of zero, use the –fill option to specify a fill value.

17.9 Controlling the ROM Device Address

The hex utility output address field corresponds to the ROM device address. The EPROM programmer burns the data in the location specified by the hex utility output file address field. The hex utility offers some mechanisms to control the starting address in ROM of each section and/or control the address index used to increment the address field. However many EPROM programmers offer direct control of where the data will be burned.

Controlling the Starting Address

The address field of the hex utility output file is controlled by the following mechanisms listed from low to high priority:

1) **The linker command file:**

By default, the address field of a hex utility output file is a function of the load address (as given in the linker command file) and the the hex utility parameter values. The relationship is summarized as follows:

$$\text{out\_file\_addr}^{\dagger} = \text{load\_addr} \times (\text{data\_width} \div \text{mem\_width})$$

out_file_addr	The address of the output file.
load_addr	The load address
data_width	Data width can be specified by the *–datawidth* command or the *datawidth* option inside the SECTIONS directive. See Section 17.3.
mem_width	The memory width of the target device. You can specify the memory width by the *–memwidth* command or the *memwidth* option inside the ROMS directive. See Section 17.3.

† If paddr is not specified

The data width/memory width fraction can be considered a correction factor for address expansion due to the fact that memory width is less than data width.

Data width can be specified by the *–datawidth* command or the *datawidth* option inside the SECTIONS directive. See Section 17.3.

Memory width can be specified by the –memwidth command or the memwidth option inside the ROMS directive. See Section 17.3.

2) **The –paddr option inside the SECTIONS directive:**

When –paddr is specified for a section, the hex utility bypasses the section load address and places the section in the address specified by –paddr. The relation between the hex utility address field and –paddr can be summarized as follows:

$$\text{out\_file}^{\dagger} = \text{paddr\_val} + (\text{load\_addr} - \text{sec\_bload}) \times (\text{data\_width} \div \text{mem\_width})$$

out_file	The address of the output file
paddr_val	The value supplied with the –paddr option inside the SECTIONS directive
sec_bload	Section beginning load address

†If paddr is specified

The data width/memory width fraction can be considered a correction factor for address expansion due to the fact that memory width is less than data width.

load address – section load address can be considered an offset from the beginning of the section.

3) **The zero option:**

When you use the –zero option, the utility resets the address origin to zero for each output file. Since each file starts at zero and counts upward, any address records represent offsets from the beginning of the file (the address within the ROM) rather than actual target addresses of the data.

You must use this option in conjunction with the –image option to force the starting address in each output file to be zero. If you specify the –zero option without the –image option, the utility issues a warning and ignores the option.

Controlling The Address Increment Index

By default, the hex utility increments the output file address field based on the value of memory width. If memory width equals 64, the address will increment based on how many words of size 64 are present in each line of the hex utility output file.

The –byte option

Some EPROM programmers may require the address field of the hex utility output file increment in a byte basis. **If you use the –byte option**, the output file address is incremented once for each byte. In an example in which the starting address is 0h and the first line contains eight 32-bit words, with –byte, the second line would start at address 32 or 020h. The data in both examples are the same; –byte affects only the calculation of the output file address field, not the actual target processor address of the converted data.

The –byte option causes the address records in an output file to refer to byte locations within the file.

Dealing With Address Holes

When memory width is different from data width, the automatic multiplication of the load address by the correction factor might create holes at the beginning of the section or in between sections.

Consider the case when you wish to load a COFF section (.sec1) at address 0x0100 of an 8-bit EPROM (one single EPROM memory system). If you specify the load address in the linker command file at location 0x100, the hex utility will multiply the address by four (data width, divided by memory width = 32/8 = 4), giving the hex utility output file starting address of 0x400. Unless you control the starting address of the EPROM in your EPROM programmer, you could create holes inside your EPROM. You can solve this issue by using the paddr command parameter inside the SECTIONS directive.

The *paddr=value* option (SECTIONS directive) forces a section to start at the provided value. Figure 17–10 shows a hex command file that can be used to avoid the hole at the beginning of .sec1. The hex conversion utility uses the address 0x100 as a starting address for .sec1 regardless of the load address assigned for this section by the linker.

Figure 17–10. Hex Command File For Hole Avoidance

```
–i
a.out
–map a.map

ROMS
{
 ROM : org = 0x0100, length = 0x200, romwidth = 8, memwidth = 8,
}

SECTIONS
 {
   sec1: paddr = 0x100
}
```

If your file contains multiple sections, and, if one section uses a paddr parameter, then all sections must use the paddr parameter.

17.10 Object Format Descriptions

The hex conversion utility converts a COFF object file into one of five object formats that most EPROM programmers accept as input: ASCII-Hex, Intel MCS-86, Motorola-S, Extended Tektronix, and TI-Tagged. This section describes each object format.

You can use one of the options in Table 17–2 to specify the hex format you want to use for the output files.

- If you use more than one of these options, the last one you list overrides the others.
- The default output format is Tektronix (–x option).

Table 17–2. Options for Specifying Hex Conversion Formats

Option	Format	Address Bits	Default Width
–a	ASCII-Hex	16	8
–i	Intel	32	8
–m	Motorola-S	16	8
–t	TI-Tagged	16	16
–x	Tektronix	32	8

Address bits determine how many bits of the address information the format supports. The formats with 16-bit addresses only support addresses up to 64K. The utility truncates target addresses to fit in the number of bits available.

The **default width** determines the default ROM width of the format. You can change the default width by using the –romwidth option or by specifying the romwidth option in the ROMS directive. You cannot change the default width of the TI-Tagged format. The TI-Tagged format supports a 16-bit width only.

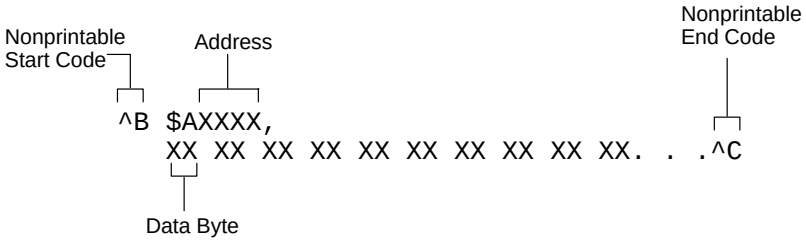
Note: Using the Intel Format

The Intel format has been extended to allow 32-bit addresses. Special linear address records (record type = 04h) are used to represent the upper 16 bits of any address requiring more than 16 bits.

17.10.1 ASCII-Hex Object Format (–a Option)

The ASCII-Hex object format supports 16-bit addresses. The format consists of a byte stream with bytes separated by spaces. Figure 17–11 illustrates the ASCII-Hex format.

Figure 17–11. ASCII-Hex Object Format



The file begins with an ASCII STX (ctrl-B, 02h) character and ends with an ASCII ETX (ctrl-C, 03h) character. Address records are indicated with $AXXXX, where XXXX is a four-digit (16-bit) hex address. The address records are present only in the following situations:

- When discontinuities occur
- When the byte stream does not begin at address 0

You can avoid all discontinuities and any address records by using the –image and –zero options. Doing this results in output that is simply a list of byte values.

17.10.2 Intel MCS-86 Object Format (–i Option)

The Intel object format supports 16-bit addresses and 32-bit extended addresses. Intel format consists of a nine-character (four-field) prefix—which defines the start of record, byte count, load address, and record type—and a two-character checksum suffix.

The three record types, which are represented in the nine-character prefix, are described below:

Record Type	Description
00	Data record
01	End-of-file record
04	Extended linear address record

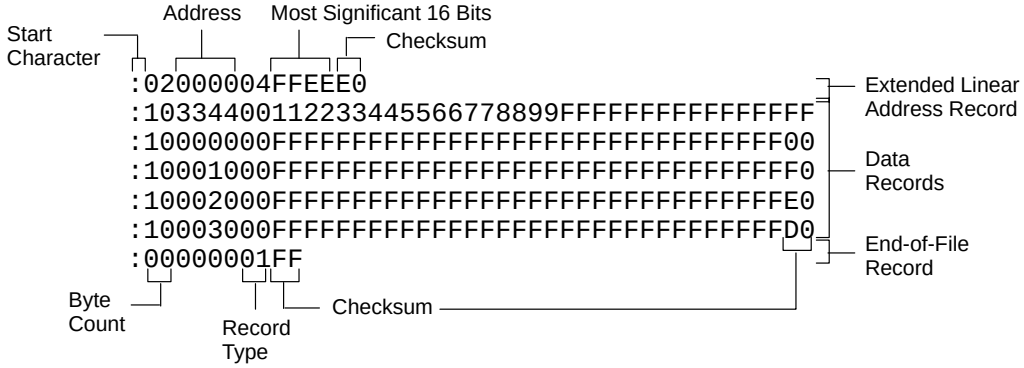
Record type *00*, the data record, begins with the colon (:) start character and is followed by the byte count, the address of the first data byte, the record type (00), and the checksum. Note that the address is the least significant 16 bits of a 32-bit address; this value is concatenated with the value from the most recent 04 (extended linear address) record to make a full 32-bit address. The checksum is the 2's complement (in binary form) of the preceding bytes in the record, including byte count, address, and data bytes.

Record type *01*, the end-of-file record, also begins with the colon (:) start character. The colon is followed by the byte count, the address, the record type (01), and the checksum.

Record type *04*, the extended linear address record, is used to specify the upper 16 address bits. It begins with the colon (:) start character. The colon is followed by the byte count, a dummy address of 0h, the record type (04), the most significant 16 bits of the address, and the checksum. The subsequent address fields in the data records have the least significant bits of the address.

Figure 17–12 illustrates the Intel hex object format.

Figure 17–12. Intel Hex Object Format



17.10.3 Motorola-S Object Format (–m Option)

The Motorola-S format supports 16-bit addresses and consists of a start-of-file (header) record, data records, and an end-of-file (termination) record. Each record is made up of five fields: record type, byte count, address, data, and checksum. The three record types are as follows:

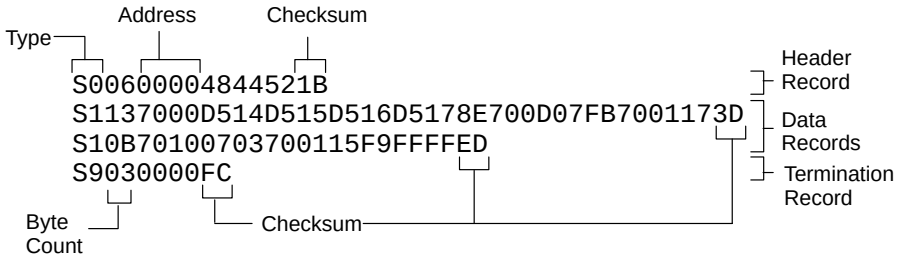
Record Type	Description
S0	Header record
S1	Code/data record
S2	Termination record

The byte count is the character pair count in the record, excluding the type and byte count itself.

The checksum is the least significant byte of the 1's complement of the sum of the values represented by the pairs of characters making up the byte count, address, and the code/data fields.

Figure 17–13 illustrates the tag characters in Motorola-S object format.

Figure 17–13. Motorola-S Format



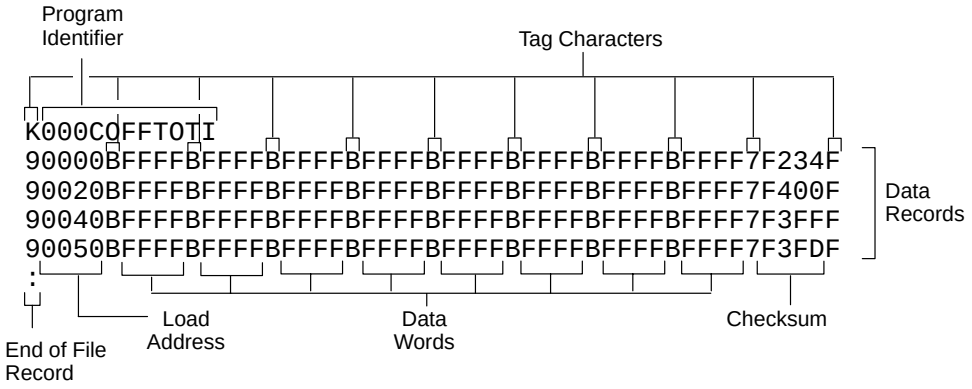
17.10.4 TI-Tagged Object Format (–t Option)

The TI-Tagged object format supports 16-bit addresses and consists of a start-of-file record, data records, and end-of-file record. Each of the data records is made up of a series of small fields and is signified by a tag character. The following is a list of the significant tag characters:

Tag Character	Description
K	followed by the program identifier
7	followed by a checksum
8	followed by a dummy checksum (ignored)
9	followed by a 16-bit load address
B	followed by a data word (four characters)
F	identifies the end of a data record
*	followed by a data byte (two characters)

Figure 17–14 illustrates the tag characters in TI-Tagged object format.

Figure 17–14. TI-Tagged Object Format



If any data fields appear before the first address, the first field is assigned address 0000h. Address fields may be expressed for any data byte, but none is required. The checksum field, which is preceded by the tag character 7, is a 2's complement of the sum of the 8-bit ASCII values of characters, beginning with the first tag character and ending with the checksum tag character (7 or 8). The end-of-file record is a (:) colon.

17.10.5 Extended Tektronix Object Format (–x Option)

The Tektronix object format supports 32-bit addresses and has two types of records: data and termination records.

termination record signifies the end of a module.

data record contains the header field, the load address, and the object code.

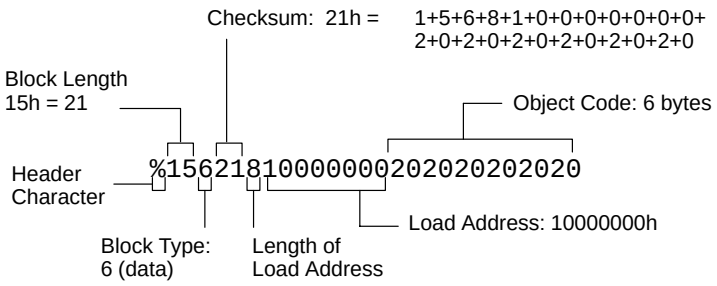
The header field in the data record contains the following information:

Item	Number of ASCII Characters	Description
%	1	Data type is Extended Tektronix hex format
Block length	2	Number of characters in the record, minus the %
Block type	1	6 = data record 8 = termination record
Checksum	2	A two-digit hex sum modulo 256 of all values in the record except the % and the checksum itself.

The load address in the data record specifies where the object code will be located. The first digit specifies the address length; this is always 8. The remaining characters of the data record contain the object code, two characters per byte.

Figure 17–15 illustrates the Tektronix object format.

Figure 17–15. Extended Tektronix Hex Object Format



17.11 Hex Conversion Utility Error Messages

1) **Sections Overlapping**

   Cause: Two or more COFF section load addresses overlap.

   Solution: This problem may be caused by an incorrect translation from load address to Hex output file address that is done by the Hex Conversion Utility when memory width is less than data width. Refer to Section 17.3, page CG:17-6.

2) **Unconfigured Memory Error**

   Cause: The COFF file contains sections outside the memory range defined in the ROMS directive.

   Solution: You should correct your ROM range as specified in your ROMS directive to cover the memory range as needed. Remember that if the ROMS directive is not used, the memory range defaults to the entire processor address space. For this reason, removing the ROMS directive could also be a feasible solution.

Chapter 18

Symbolic Debugging Directives

The TMS320C8x assemblers support several directives that the TMS320C8x C compiler and debugger use for symbolic debugging. These symbolic debugging directives are not usually listed in the assembly language file that the compiler creates. If you want them to be listed, invoke the code generator with the –g option, as shown below:

```
ppcl –g input_file or
mpcl –g input_file
```

This chapter contains an alphabetical directory of the symbolic debugging directives. Each listing contains an example of C source and the resulting assembly language code.

Syntax

.block *beginning line number*

.endblock *ending line number*

Description

The .block and .endblock directives specify the beginning and end of a C block. The *line numbers* are optional; they specify the location in the source file where the block is defined.

Note that block definitions can be nested. The assembler will detect improper block nesting.

Example

Here is an example of C source that defines a block, and the resulting assembly language code.

C source:

```
     .
     .
     .
   {                    /*  Beginning of a block */
        int  a,b;
        a = b;
   }                    /*  End of a block       */
     .
     .
     .
```

Resulting assembly language code:

```
        d0 = &*(sp -–= 8)
        .block 5
        .sym $a,0,4,1,32
        .sym b,4,4,1,32
        d1 =sw *(sp + 4)
        *(sp + 0) =w d1
        .endblock 7
        br = iprs
        nop
        d0 = &*(sp ++= 8)
;       branch occurs here
```

Syntax **.file** *”filename”*

Description The .file directive allows a debugger to map locations in memory back to lines in a C source file. The *filename* is the name of the file that contains the original C source program. The first 14 characters of the filename are significant.

You can also use the .file directive in assembly code to provide a name in the file and improve program readability.

Example Here is an example of the .file directive. The file named *text.c* contains the C source that produced this directive.

```
.file    ”text.c”
```

Syntax

.func *beginning line number*

.endfunc *ending line number*

Description

The .func and .endfunc directives specify the beginning and end of a C function. The *line numbers* are optional; they specify the location in the source file where the function is defined.

Note that function definitions cannot be nested.

Example

Here is an example of C source that defines a function, and the resulting assembly language code.

C source:

```
power(x, n)  /*  Beginning of a function */
int x,n;
{
   int i, p;
   p = 1;
   for (i = 1; i <= n; ++i)
      p = p * x;
   return p; /*  End of function         */
}
```

Resulting assembly language code:

```
        .func 2
$power:
        d0 = &*(sp --= 16)
;* d1    assigned to $x
        .sym $x,1,4,17,32
;* d2    assigned to $n
        .sym $n,2,4,17,32
        .sym $x,0,4,1,32
        .sym $n,4,4,1,32
        .sym $i,8,4,1,32
        .sym $p,12,4,1,32
        .line 3
        *(sp + 4) =w d2
||        *(sp + 0) =w d1
        .line 5
        d1 = 1
        *(sp + 12) =w d1
        .line 6
        d1 = 1
        *(sp + 8) =w d1
        d2 =sw *(sp + 4)
||        d1 =sw *(sp + 8)
        d1 = d1 - d2
        br =[gt] L3
        nop
        nop
;       branch occurs here
L2:
        .line 7
        d1 =sw *(sp + 12)
        d3 =uh1 d1
||        d2 =sw *(sp + 0)
        d4 =u d2 * d3
||        d3 =uh1 d2
        d3 =u d3 * d1
        d1 =u d2 * d1
||        d2 = d3 + d4
        d1 = d1 + (d2 << 16)
        *(sp + 12) =w d1
        .line 6
        d1 =sw *(sp + 8)
        d1 = d1 + 1
        *(sp + 8) =w d1
        d2 =sw *(sp + 4)
||        d1 =sw *(sp + 8)
        d1 = d1 - d2
        br =[le] L2
        nop
        nop
;       branch occurs here
L3:
        .line 8
        d5 =sw *(sp + 12)
        .line 9
        br = iprs
        nop
        d0 = &*(sp ++= 16)
;       branch occurs here
        .endfunc 10,0x00000000,16
```

Syntax

.line *line number* [**,** *address*]

Description

The .line directive creates a line number entry in the object file. Line number entries are used in symbolic debugging to associate addresses in the object code with the lines in the source code that generated them.

The .line directive has two operands:

- *Line number* indicates the line of the C source that generated a portion of code. Line numbers are relative to the beginning of the current function. This is a required parameter.
- *Address* is an expression that is the address associated with the line number. This is an optional parameter; if you don't specify an address, the assembler will use the current SPC value.

Example

The .line directive is followed by the assembly language source statements that are generated by the indicated line of C source. For example, assume that the lines of C source below are lines 4 and 5 in the original C source; lines 5 and 6 produce the assembly language source statements that are shown below.

C source:

```
for (i = 1; i <= n; ++i)
     p = p * x;
```

Resulting assembly language code:

```
;*************************************************************
;* FUNCTION DEF: $main                                       *
;*************************************************************
$main:
        d0 = &*(sp --= 12)
        .sym $i,0,4,1,32
        .sym $n,4,4,1,32
        .sym $x,8,4,1,32
        .line 4
        d1 = 1
        *(sp + 0) =w d1

        d2 =sw *(sp + 4)
||        d1 =sw *(sp + 0)
        d1 = d1 – d2
        br =[gt] L3
        nop
        nop
;       branch occurs here
L2:
        .line 5
        d1 =sw *(sp + 8)

        d3 =uh1 d1
||        d2 =sw *(sp + 0)

        d3 =u d2 * d3
||        d4 =uh1 d2
        d4 =u d4 * d1

        d1 =u d2 * d1
||        d2 = d4 + d3
        d1 = d1 + (d2 << 16)
        *(sp + 8) =w d1
        .line 4
        d1 =sw *(sp + 0)
        d1 = d1 + 1
        *(sp + 0) =w d1

        d2 =sw *(sp + 4)
||        d1 =sw *(sp + 0)
        d1 = d1 – d2
        br =[le] L2
        nop
        nop
;       branch occurs here
L3:
        .line 6
        br = iprs
        nop
        d0 = &*(sp ++= 12)
;       branch occurs here
        .endfunc 7,0x00000000,12
```

Syntax

.member *name*, *value* [, *type*, *storage class*, *size*, *tag*, *dims*]

Description

The .member directive defines a member of a structure, union, or enumeration. It is valid only when it appears in a structure, union, or enumeration definition.

name	is the name of the member that is put in the symbol table. The first 32 characters of the name are significant.
value	is the value associated with the member. Any legal expression (absolute or relocatable) is acceptable.
type	is the C type of the member.
storage class	is the C storage class of the member
size	is the number of bits of memory required to contain this member.
tag	is the name of the type (if any) or structure of which this member is a type. This name **must** have been previously declared by a .stag, .etag, or .utag directive.
dims	may be one to four expressions separated by commas. This allows up to four dimensions to be specified for the member.

The order of parameters is significant. *Name* and *value* are required parameters. All other parameters may be omitted or empty. (Adjacent commas indicate an empty entry.) This allows you to skip a parameter and specify a parameter that occurs later in the list. Operands that are omitted or empty assume a null value.

Example

Here is an example of a C structure definition and the corresponding assembly language statements:

C source:

```
struct doc {
     char  title;
     char  group;
     int   job_number;
}  doc_info;
```

Resulting assembly language code:

```
      .stag   doc,48
      .member $title,0,2,8,8
      .member $group,8,2,8,8
      .member $job_number,16,4,8,32
      .eos
```

Syntax

.**stag** *name* [**,** *size*]
member definitions
.etag *name* [**,** *size*]
member definitions
.utag *name* [**,** *size*]
member definitions
.eos

Description

The .stag directive begins a structure definition. The .etag directive begins an enumeration definition. The .utag directive begins a union definition. The .eos directive ends a structure, enumeration, or union definition.

name	is the name of the structure, enumeration, or union. The first 32 characters of the name are significant. This is a required parameter.
size	is the number of bits the structure, enumeration, or union occupies in memory. This is an optional parameter; if omitted, the size is unspecified.

The .stag, .etag, or .utag directive should be followed by a number of .member directives, which define members in the structure. The .member directive is the only directive that can appear inside a structure, enumeration, or union definition.

The assembler does not allow nested structures, enumerations, or unions. The C compiler unwinds nested structures by defining them separately and then referencing them from the structure they are referenced in.

Example 1 Here is an example of a structure definition.

C source:

```
struct doc
{
      char  title;
      char  group;
      intjob_number;
}  doc_info;
```

Resulting assembly language code:

```
         .stag $doc,64
         .member  $title,0,2,8,8
         .member  $group,8,2,8,8
         .member  $job_number,32,4,8,32
         .eos
         .global  $doc_info
         .global  $doc_info
$doc_info:   .usect.pbss,8,4
         .sym  $doc_info,$doc_info,8,2,64,$doc
```

Example 2 Here is an example of a union definition.

C source:

```
union u_tag {
     int   val1;
     float val2;
     char  valc;
}  valu;
```

Resulting assembly language code:

```
   .utag $u_tag,32
   .member  $val1,0,4,11,32
   .member  $val2,0,4,11,32
   .member  $valc,0,4,11,32
   .eos
```

Example 3 Here is an example of an enumeration definition.

C Source:

```
{
    enum o_ty { reg_1, reg_2, result } optypes;
}
```

Resulting assembly language code:

```
   .etag $o_ty,32
   .member  $reg_1,0,4,16,32
   .member  $reg_2,1,4,16,32
   .member  $result,2,4,16,32
   .eos
```

Syntax

.sym *name*, *value* [, *type*, *storage class*, *size*, *tag*, *dims*]

Description

The .sym directive specifies symbolic debugging information about a global variable, local variable, or a function.

name	is the name of the variable that is put in the object symbol table. The first 32 characters of the name are significant.
value	is the value associated with the variable. Any legal expression (absolute or relocatable) is acceptable.
type	is the C type of the member.
storage class	is the C storage class of the member
size	is the number of bits of memory required to contain this member.
tag	is the name of the type (if any) or structure of which this member is a type. This name **must** have been previously declared by a .stag, .etag, or .utag directive.
dims	may be one to four expressions separated by commas. This allows up to four dimensions to be specified for the member.

The order of parameters is significant. The *name* and *value* are required parameters. All other parameters may be omitted or empty (adjacent commas indicate an empty entry). This allows you to skip a parameter and specify a parameter that occurs later in the list. Operands that are omitted or empty assume a null value.

Example

These lines of C source produce the .sym directives shown below:

C source:

```
struct s { int member1, member2; } str;
int     ext;
int     array[5][10];
long  *ptr;
int     strcmp();

main(arg1,arg2)
   int   arg1;
   char  *arg2;
{
   register r1;
}
```

Resulting assembly language code:

```
;* d1    assigned to $arg1
   .sym   $arg1,1,4,17,32
;* d2    assigned to $arg2
   .sym   $arg2,2,18,17,32
   .sym   $arg1,0,4,1,32
   .sym   $arg2,4,18,1,32
;* d1    assigned to $r1
   .sym   $r1,1,4,4,32
```

Appendix A

Common Object File Format

The assembler and linker create object files in common object file format (COFF). COFF is an implementation of an object file format of the same name that was developed by AT&T for use on UNIX-based systems. This format is used because it encourages modular programming and provides more powerful and flexible methods for managing code segments and target system memory.

Sections are a basic COFF concept. Chapter 12, *Introduction to Common Object File Format* discusses COFF sections in detail. If you understand section operation, you will be able to use the code generation tools more efficiently.

This appendix contains technical details about COFF object file structure. Much of this information pertains to the symbolic debugging information that is produced by the C compiler. The purpose of this appendix is to provide supplementary information on the internal format of COFF object files.

Topics

A.1 COFF File Structure

The elements of a COFF object file describe the file's sections and symbolic debugging information. These elements include:

- A file header
- Optional header information
- A table of section headers
- Raw data for each initialized section
- Relocation information for each initialized section
- Line number entries for each initialized section
- A symbol table
- A string table

The assembler and linker produce object files with the same COFF structure; however, a program that is linked for the final time does not usually contain relocation entries. Figure A–1 illustrates the overall object file structure.

Figure A–1. COFF File Structure

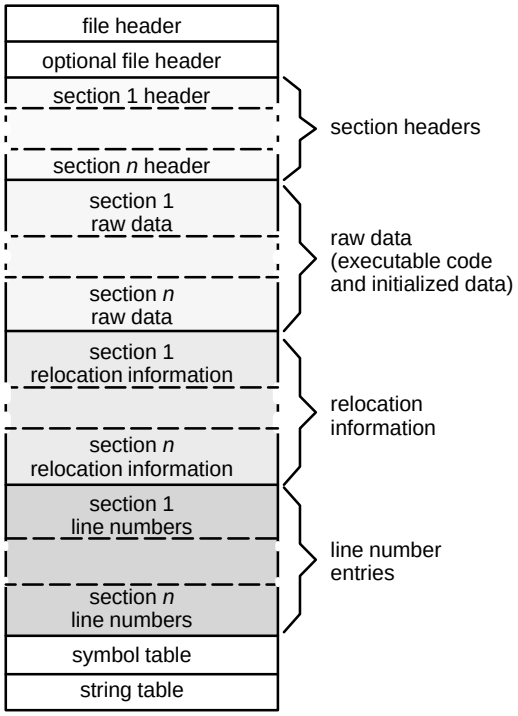
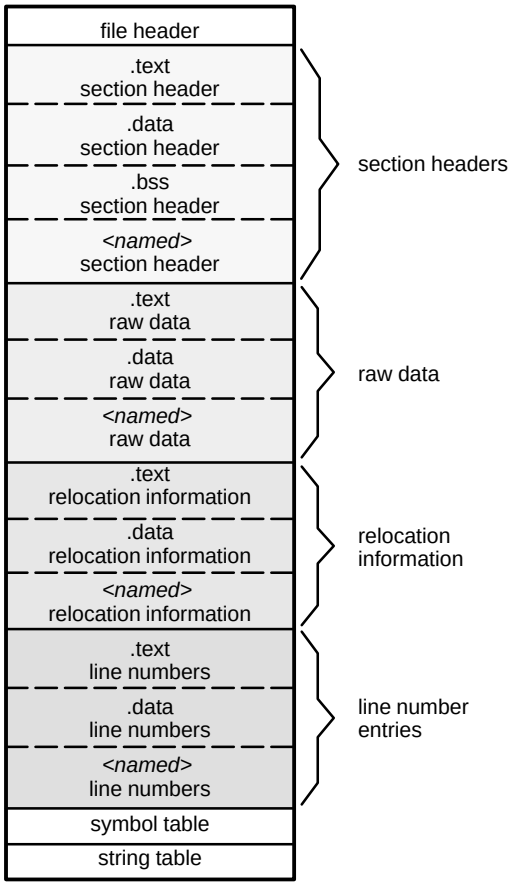


Figure A–2 shows a typical example of a COFF object file that contains the three default sections—.text, .data, and .bss—and a named section (referred to as <named>). Although uninitialized sections have section headers, they have no raw data, relocation information, or line number entries. This is because the .bss and .usect directives simply reserve space for uninitialized data; uninitialized sections contain no actual code.

Figure A–2. Sample COFF Object File



A.2 File Header Structure

The file header contains information that describes the general format of an object file. Table A–1 shows the structure of the COFF file header (22 bytes).

Table A–1. File Header Contents for COFF

Byte Number	Type	Description
0–1	Unsigned short integer	Version id (0c1h); indicates version of COFF file structure.
2–3	Unsigned short integer	Number of section headers
4–7	Long integer	Time and date stamp; indicates when the file was created
8–11	Long integer	File pointer; contains the symbol table's starting address
12–15	Long integer	Number of entries in the symbol table
16–17	Unsigned short integer	Number of bytes in the optional header. This field is either 0 or 28; if it is 0, there is no optional file header.
18–19	Unsigned short integer	Flags (see Table A–2)
20–21	Unsigned short integer	Target id; magic number (095h) indicates the file can be executed in a 'C8x system.

Table A–2 lists the flags that can appear in bytes 18 and 19 of the file header. Any number and combination of these flags can be set at the same time (for example, if bytes 18 and 19 are set to 0003h, F_RELFLG and F_EXEC are both set.)

Table A–2. File Header Flags (Bytes 18 and 19)

Mnemonic	Flag	Description
F_RELFLG	0001h	Relocation information was stripped from the file.
F_EXEC	0002h	The file is executable (it contains no unresolved external references).
F_LNNO	0004h	Line numbers were stripped from the file.
F_LSYMS	0008h	Local symbols were stripped from the file.
F_LITTLE	0100h	The file has little endian byte ordering.
F_BIG	0200h	The file has big endian byte ordering.

A.3 Optional File Header Format

The linker creates the optional file header and uses it to perform relocation at download time. Partially linked files do not contain optional file headers. Table A–3 illustrates the optional file header format.

Table A–3. Optional File Header Contents

Byte Number	Type	Description
0–1	Short integer	Optional file header magic number (108h)
2–3	Short integer	Version stamp
4–7	Long integer	Size (in bytes) of executable code
8–11	Long integer	Size (in bytes) of initialized data
12–15	Long integer	Size (in bytes) of uninitialized data
16–19	Long integer	Entry point
20–23	Long integer	Beginning address of executable code
24–27	Long integer	Beginning address of initialized data

A.4 Section Header Structure

COFF object files contain a table of section headers that define where each section begins in the object file. Each section has its own section header.

Table A–4. Section Header Contents

Byte Number	Type	Description
0–7	Character	8-character section name, padded with nulls
8–11	Long integer	Section's physical address
12–15	Long integer	Section's virtual address
16–19	Long integer	Section size in words
20–23	Long integer	File pointer to raw data
24–27	Long integer	File pointer to relocation entries
28–31	Long integer	File pointer to line number entries
32–33	Unsigned short integer	Number of relocation entries
34–35	Unsigned short integer	Number of line number entries
36–37	Unsigned short integer	Flags (see Table A–5)
38	Character	Reserved
39	Unsigned character	Memory page number

Table A–5 lists the flags that can appear in bytes 36 and 37 of the section header.

Table A–5. Section Header Flags (Bytes 36 and 37)

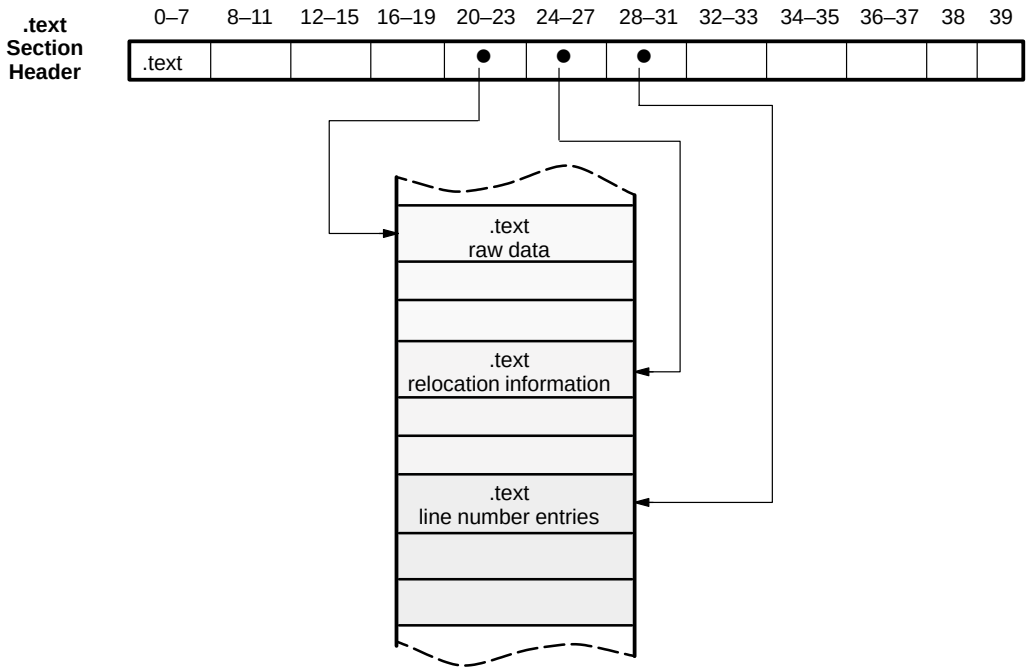
Mnemonic	Flag	Description
STYP_REG	0000h	Regular section (allocated, relocated, loaded)
STYP_DSECT	0001h	Dummy section (relocated, not allocated, not loaded)
STYP_NOLOAD	0002h	Noload section (allocated, relocated, not loaded)
STYP_GROUP	0004h	Grouped section (formed from several input sections)
STYP_PAD	0008h	Padding section (loaded, not allocated, not relocated)
STYP_COPY	0010h	Copy section (relocated, loaded, but not allocated; relocation and line number entries are processed normally)
STYP_TEXT	0020h	Section contains executable code
STYP_DATA	0040h	Section contains initialized data
STYP_BSS	0080h	Section contains uninitialized data
STYP_PASS	2000h	Pass the section through unchanged

Note: The term *loaded* means that the raw data for this section appears in the object file.

The flags listed in Table A–5 can be combined; for example, if the flag's word is set to 024h, both STYP_GROUP and STYP_TEXT are set.

Example A–1 illustrates how the pointers in a section header would point to the elements in an object file that are associated with the .text section.

Example A–1. Section Header Pointers for the .text Section



As Figure A–2 on page CG:A-3, shows uninitialized sections (created with the .bss and .usect directives) vary from this format. Although uninitialized sections have section headers, they have no raw data, relocation information, or line number information. They occupy no actual space in the object file. Therefore, the number of relocation entries, the number of line number entries, and the file pointers are 0 for an uninitialized section. The header of an uninitialized section simply tells the linker how much space for variables it should reserve in the memory map.

A.5 Structuring Relocation Information

A COFF object file has one relocation entry for each relocatable reference. The assembler automatically generates relocation entries. The linker reads the relocation entries as it reads each input section and performs relocation. The relocation entries determine how references within each input section are treated.

COFF file relocation information entries use the 12-byte format shown in Table A–6.

Table A–6. Relocation Entry Contents for COFF

Byte Number	Type	Description
0–3	Long integer	Virtual address of the reference
4–7	Unsigned long integer	Symbol table index
8–9	Unsigned short integer	Reserved
10–11	Unsigned short integer	Relocation type (see Table A–7)

The **virtual address** is the symbol's address in the current section *before* relocation; it specifies *where* a relocation must occur. (This is the address of the field in the object code that must be patched.)

Following is an example of PP code that generates a relocation entry:

```
1                                       .global X
2 00000000 97801BC000000000!           call = X
```

In this example, the virtual address of the relocatable field is 00000000h.

The **symbol table index** is the index of the referenced symbol. In the preceding example, this field would contain the index of X in the symbol table. The size of the relocation is the difference between the symbol's relocation address in the section and its assembly-time address. The relocatable field must be relocated by the same amount as the referenced symbol. In the example, X has a value of 0 before relocation. Suppose X is relocated to address 02000000h. This is the relocation amount (02000000h – 0 = 2000000h), so the relocation field at address 0 is patched by adding 2000h to it.

You can determine a symbol's relocated address if you know which section it is defined in. For example, if X is defined in .data and .data is relocated by 02000000h, X is relocated by 02000000h.

If the symbol table index in a relocation entry is –1 (0FFFFFFFFh), this is called an *internal relocation.* In this case, the relocation amount is simply the amount by which the current section is being relocated.

The **relocation type** specifies the size of the field to be patched and describes how the patched value should be calculated. The type field depends on the instruction generated the relocatable reference. In the preceding example, the actual address of the referenced symbol X will be placed in a 32-bit field in the object code. This is a 32-bit address that will be accessed directly, so the relocation type is R_RELLONG. Table A–7 lists the relocation types.

Table A–7. Relocation Types (Bytes 10 and 11)

Mnemonic	Flag	Relocation Type
R_ABS	0000h	absolute address – no relocation
R_RELLONG	011h	PP: 32 bits, direct
R_PPBASE	0034h	PP: Global Base address type
R_PPLBASE	0035h	PP: Local Base address type
R_PP15	0038h	PP: Global 15 bit offset
R_PP15W	0039h	PP: Global 15 bit offset divided by 4
R_PP15H	003Ah	PP: Global 15 bit offset divided by 2
R_PP16B	003Bh	PP: Global 16 bit offset for bytes
R_PPL15	003Ch	PP: Local 15 bit offset
R_PPL15W	003Dh	PP: Local 15 bit offset divided by 4
R_PPL15H	003Eh	PP: Local 15 bit offset divided by 2
R_PPL16B	003Fh	PP: Local 16 bit offset for bytes
R_PPN15	0040h	PP: Global 15 bit negative offset
R_PPN15W	0041h	PP: Global 15 bit negative offset / 4
R_PPN15H	0042h	PP: Global 15 bit negative offset / 2
R_PPN16B	0043h	PP: Global 16 bit negative byte offset
R_PPLN15	0044h	PP: Local 15 bit negative offset
R_PPLN15W	0045h	PP: Local 15 bit negative offset / 4
R_PPLN15H	0046h	PP: Local 15 bit negative offset / 2
R_PPLN16B	0047h	PP: Local 16 bit negative byte offset
R_MPPCR	004Fh	MP: 32-bit PC-relative / 4

A.6 Line Number Table Structure

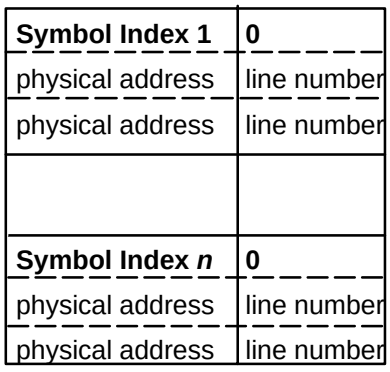
The object file contains a table of line number entries that are useful for symbolic debugging. When the C compiler produces several lines of assembly language code, it creates a line-number entry that maps these lines back to the original line of C source code that generated them. Each single line number entry contains 6 bytes of information. Table A–8 shows the format of a line-number entry.

Table A–8. Line Number Entry Format

Byte Number	Type	Description
0–3	Long integer	This entry may have one of two values: 1) If it is the first entry in a block of line-number entries, it points to a symbol entry in the symbol table. 2) If it is not the first entry in a block, it is the physical address of the line indicated by bytes 4–5.
4–5	Unsigned short integer	This entry may have one of two values: 1) If this field is 0, this is the first line of a function entry. 2) If this field is *not* 0, this is the line number of a line of C source code.

Figure A–3 shows how line number entries are grouped into blocks.

Figure A–3. Line Number Blocks



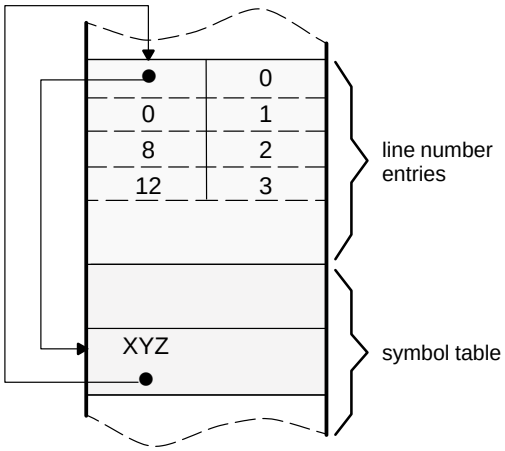
As Figure A–3 shows, each entry is divided into halves:

- For the *first line* of a function, bytes 0–3 point to the name of a symbol or a function in the symbol table, and bytes 4–5 contain a 0, which indicates the beginning of a block.
- For the *remaining lines* in a function, bytes 0–3 show the physical address (the number of bytes created by a line of C source), and bytes 4–5 show the address of the original C source, relative to its appearance in the C source program.

The line entry table can contain many of these blocks.

Figure A–4 illustrates line number entries for a function named XYZ. As shown, the function name is entered as a symbol in the symbol table. The first portion on XYZ's block of line number entries points to the function name in the symbol table. Assume that the original function in the C source contained three lines of code. The code associated with the first line is located at byte offset 0 from the beginning of the function. The code for line 2 begins at offset 8, and the code associated with line 3 is 12 bytes from the beginning of the function.

Figure A–4. Line Number Entries



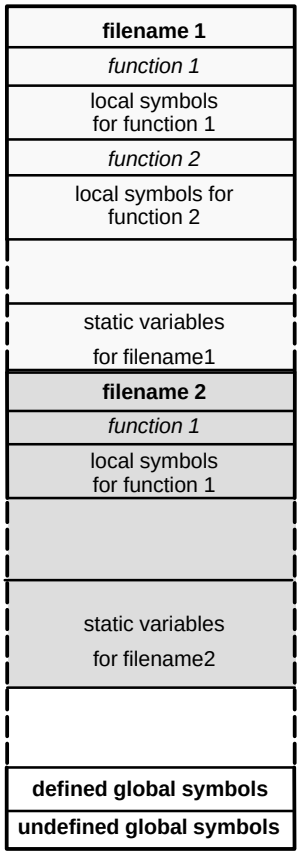
(Note that the symbol table entry for XYZ has a field that points back to the beginning of the line number block.)

Because line numbers are not often needed, the linker provides an option (–s) that strips line number information from the object file; this provides a more compact object module.

A.7 Symbol Table Structure and Content

The order of symbols in the symbol table is shown in Figure A–5.

Figure A–5. Symbol Table Contents



A *static* symbol is a symbol with scope confined to the module that defines it. This includes symbols defined in C with the static storage class outside of any function, and some assembly symbols such as section symbol names. If you have several modules that use symbols with the same name, making them static confines the scope of each symbol to the module that defines it (this eliminates multiple-definition conflicts).

The entry for each symbol in the symbol table contains the symbol's:

- Name (or an offset into the string table)
- Type
- Value
- Section it was defined in
- Storage class
- Basic type (integer, character, etc.)
- Derived type (array, structure, etc.)
- Dimensions
- Line number of the source code that defined the symbol.

Section names are also defined in the symbol table.

All symbol entries, regardless of class and type, have the same format in the symbol table. Each symbol table entry contains the 18 bytes of information listed in Table A–9. Each symbol may also have an 18-byte auxiliary entry; the special symbols listed in Table A–10 always have an auxiliary entry. Some symbols may not have all the characteristics listed above.

Table A–9. Symbol Table Entry Contents

Byte Number	Type	Description
0–7	Character	This field contains one of the following: 1) An 8-character symbol name, padded with nulls 2) An offset into the string table if the symbol name is longer than 8 characters
8–11	Long integer	Symbol value; storage class dependent
12–13	Short integer	Section number of the symbol
14–15	Unsigned short integer	Basic and derived type specification
16	Character	Storage class of the symbol
17	Character	Number of auxiliary entries (always 0 or 1)

A.7.1 Special Symbols

The symbol table contains some special symbols that are generated by the compiler, assembler, and linker. Each special symbol contains ordinary symbol table information and a possible auxiliary entry. Table A–10 lists these symbols.

Table A–10. Special Symbols in the Symbol Table

Symbol	Description
.file	File name
.text	Address of the .text section
.data	Address of the .data section
.bss	Address of the .bss section
.bb	Address of the beginning of a block
.eb	Address of the end of a block
.bf	Address of the beginning of a function
.ef	Address of the end of a function
.target	Pointer to a structure or union that is returned by a function
.tg*n*	Dummy tag name for a structure, union, or enumeration
.eos	End of a structure, union, or enumeration
etext	Next available address after the end of the .text output section
edata	Next available address after the end of the .data output section
end	Next available address after the end of the .bss output section

Several of these symbols appear in pairs:

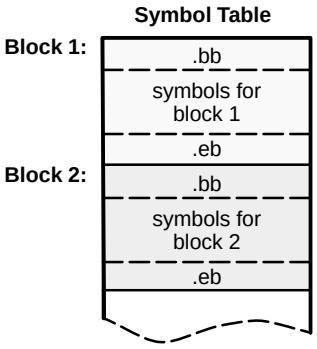
- .bb/.eb indicate the beginning and end of a block.
- .bf/.ef indicate the beginning and end of a function.
- .tg*n*/.eos or .fake*n*/.eos name and define the limits of structures, unions, and enumerations that were not named. The .eos symbol is also paired with named structures, unions, and enumerations.

When a structure, union, or enumeration has no tag name, the compiler assigns it a name so that it can be entered into the symbol table. These names are of the form .tg*n*, where *n* is an integer. The compiler begins numbering these symbol names at 0.

Symbols and Blocks

In C, a block is a compound statement that begins and ends with braces. A block always contains symbols. The symbol definitions for any particular block are grouped together in the symbol table and are delineated by the .bb/.eb special symbols. Note that blocks can be nested in C, and their symbol table entries can be nested correspondingly. Figure A–6 shows how block symbols are grouped in the symbol table.

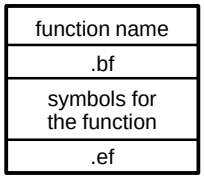
Figure A–6. Symbols for Blocks



Symbols and Functions

The symbol definitions for a function appear in the symbol table as a group, delineated by .bf/.ef special symbols. The symbol table entry for the function name precedes the .bf special symbol. Section 1.1.2, *Specifying Filenames*, shows the format of symbol table entries for a function.

Figure A–7. Symbols for Functions



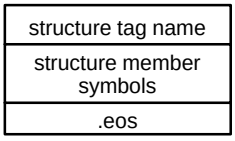
If a function returns a structure or union, a symbol table entry for the special symbol .target will appear between the entries for the function name and the .bf special symbol.

Symbols and Structures, Unions, and Enums

The symbol definitions for a structure, union, or enumerated type appears in the symbol table as a group bounded by the structure

tag name and the .eos symbols. Figure A–8 shows the format of these symbol table entries.

Figure A–8. Symbols for Structures, Unions, and Enumerated Data Types



A.7.2 Symbol Name Format

The first 8 bytes of a symbol table entry (bytes 0–7) indicate a symbol's name:

- If the symbol name is 8 characters or less, this field has type *character.* The name is padded with nulls (if necessary) and stored in bytes 0–7.
- If the symbol name is greater than 8 characters, this field is treated as two long integers. The entire symbol name is stored in the string table. Bytes 0–3 contain 0, and bytes 4–7 are an offset into the string table.

A.7.3 String Table Structure

Symbol names that are longer than eight characters are stored in the string table. The field in the symbol table entry that would normally contain the symbol's name contains, instead, a pointer to the symbol's name in the string table. Names are stored contiguously in the string table, delimited by a null byte. The first four bytes of the string table contain the size of the string table in bytes; thus, offsets into the string table are greater than or equal to four.

Figure A–9 shows an example of a string table that contains two symbol names, Adaptive-Filter and Fourier-Transform. The index in the string table is 4 for Adaptive-Filter and 20 for Fourier-Transform.

Figure A–9. Sample String Table

38			
'A'	'd'	'a'	'p'
't'	'i'	'v'	'e'
'-'	'F'	'i'	'l'
't'	'e'	'r'	'\0'
'F'	'o'	'u'	'r'
'i'	'e'	'r'	'-'
'T'	'r'	'a'	'n'
's'	'f'	'o'	'r'
'm'	'\0'		

A.7.4 Storage Classes

Byte 16 of the symbol table entry indicates the storage class of the symbol. Storage classes refer to the method in which the C compiler accesses a symbol. Table A–11 lists valid storage classes.

Table A–11. Symbol Storage Classes

Some special symbols are restricted to certain storage classes. Table A–12 lists these symbols and their storage classes.

Mnemonic	Value	Storage Class	Mnemonic	Value	Storage Class
C_NULL	0	No storage class	C_USTATIC	14	Undefined static
C_AUTO	1	Automatic variable	C_ENTAG	15	Enumeration tag
C_EXT	2	External definition	C_MOE	16	Member of an enumeration
C_STAT	3	Static	C_REGPARM	17	Register parameter
C_REG	4	Register variable	C_FIELD	18	Bit field
C_EXTREF	5	External reference	C_UEXT	19	Tentative external definition
C_LABEL	6	Label	C_STATLAB	20	Static load time label
C_ULABEL	7	Undefined label	C_EXTLAB	21	External load time label
C_MOS	8	Member of a structure	C_SYSTEM	23	System Wide Variable
C_ARG	9	Function argument	C_BLOCK	100	Beginning or end of a block; used only for the .bb and .eb special symbols
C_STRTAG	10	Structure tag	C_FCN	101	Beginning or end of a function; used only for the .bf and .ef special symbols
C_MOU	11	Member of a union	C_EOS	102	End of structure; used only for the .eos special symbol
C_UNTAG	12	Union tag	C_FILE	103	Filename; used only for the .file special symbol
C_TPDEF	13	Type definition	C_LINE	104	Used only by utility programs

Table A–12. Special Symbols and Their Storage Classes

Special Symbol	Restricted to This Storage Class	Special Symbol	Restricted to This Storage Class
.file	C_FILE	.eos	C_EOS
.bb	C_BLOCK	.text	C_STAT
.eb	C_BLOCK	.data	C_STAT
.bf	C_FCN	.bss	C_STAT
.ef	C_FCN		

A.7.5 Symbol Values

Bytes 8–11 of a symbol table entry indicate a symbol's value. A symbol's value depends on the symbol's storage class; Table A–13 summarizes the storage classes and related values.

Table A–13. Symbol Values and Storage Classes

Storage Class	Value Description	Storage Class	Value Description
C_AUTO	Stack offset in bytes	C_UNTAG	0
C_EXT	Relocatable address	C_TPDEF	0
C_SYSTEM	Relocatable address	C_ENTAG	0
C_STAT	Relocatable address	C_MOE	Enumeration value
C_REG	Register number	C_REGPARM	Register number
C_LABEL	Relocatable address	C_FIELD	Bit displacement
C_MOS	Offset in bits	C_BLOCK	Relocatable address
C_ARG	Stack offset in bytes	C_FCN	Relocatable address
C_STRTAG	0	C_FILE	0
C_MOU	Offset in bits		

If a symbol's storage class is C_FILE, the symbol's value is 0 in an unlinked file, and a pointer to the next .file symbol in a linked file. The .file symbols form a one-way linked list in the symbol ta-

ble. The last .file entry points to the first global symbol in the symbol table.

The value of a relocatable symbol is its virtual address. When the linker relocates a section, the value of a relocatable symbol changes accordingly.

A.7.6 Section Number

Bytes 12–13 of a symbol table entry contain a number that indicates which section the symbol was defined in. Table A–14 lists these numbers and the sections they indicate.

Table A–14. Section Numbers

Mnemonic	Section Number	Description
N_DEBUG	–2	Special symbolic debugging symbol
N_ABS	–1	Absolute symbol
N_UNDEF	0	Undefined external symbol
N_SCNUM	1	.text section (typical)
N_SCNUM	2	.data section (typical)
N_SCNUM	3	.bss section (typical)
N_SCNUM	1–32,767	Section number of a named section, in the order in which the named sections are encountered

Note that if there were no .text, .data, or .bss sections, the numbering of named sections would begin with 1.

If a symbol has a section number of 0, –1, or –2, it is not defined in a section. A section number of –2 indicates a symbolic debugging symbol, which includes structure, union, and enumeration tag names, type definitions, and the filename. A section number of –1 indicates that the symbol has a value but is not relocatable. A section number of 0 indicates a relocatable external symbol that is not defined in the current file.

A.7.7 Type Entry

Bytes 14–15 of the symbol table entry define the symbol's type. Each symbol has one basic type and one to six derived types.

Following is the format for this 16-bit type entry:

	Derived Type 6	Derived Type 5	Derived Type 4	Derived Type 3	Derived Type 2	Derived Type 1	Basic Type
Size (in bits):	2	2	2	2	2	2	4

Bits 0–3 of the type field indicate the basic type. Table A–15 lists valid basic types.

Table A–15. Basic Types

Mnemonic	Value	Type
T_VOID	0	Void type
T_SCHAR1	1	Character (explicitly signed)
T_CHAR	2	Character (implicitly signed)
T_SHORT	3	Short integer
T_INT	4	Integer
T_LONG	5	Long integer
T_FLOAT	6	Floating point
T_DOUBLE	7	Double floating point
T_STRUCT	8	Structure
T_UNION	9	Union
T_ENUM	10	Enumeration
T_LDOUBLE	11	Long Double Floating Point
T_UCHAR	12	Unsigned character
T_USHORT	13	Unsigned short integer
T_UINT	14	Unsigned integer
T_ULONG	15	Unsigned long integer

Bits 4–15 of the type field are arranged as six 2-bit fields that can indicate one to six derived types. Table A–16 lists the possible derived types.

Table A–16. Derived Types

Mnemonic	Value	Type
DT_NON	0	No derived type
DT_PTR	1	Pointer
DT_FCN	2	Function
DT_ARY	3	Array

An example of a symbol with several derived types would be a symbol with a type entry of 0000000001110100_2. This entry indicates that the symbol is an array of pointers to integers.

A.7.8 Auxiliary Entries

Each symbol table entry may have **one** or **no** auxiliary entry. An auxiliary symbol table entry contains the same number of bytes as a symbol table entry (18), but the format of an auxiliary entry depends on the symbol's type and storage class. Table A–17 summarizes these relationships. A symbol that satisfies none of these conditions should not have an auxiliary entry.

Table A–17. Auxiliary Symbol Table Entries Format

Name	Storage Class	Type Entry		Auxiliary Entry Format
		Derived Type 1	Basic Type	
.file	C_FILE	DT_NON	T_VOID	Filename (see Table A–18)
section name	C_STAT	DT_NON	T_VOID	Section (see Table A–19)
structure tag name	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_STRUCT T_UNION T_ENUM	Tag name (see Table A–20)
.eos	C_EOS	DT_NON	T_VOID	End of structure (see Table A–21)
function name	C_EXT C_STAT	DT_FCN	(Any)	Function (see Table A–22)
array name	C_AUTO, C_STAT, C_MOS, C_MOU, C_TPDEF, C_EXT	DT_ARY	Any except T_VOID	Array (see Table A–23)
.bb, .eb	C_BLOCK	DT_NON	T_VOID	Beginning and end of a block (see Table A–24 and Table A–25)
.bf, .ef	C_FCN	DT_NON	T_VOID	Beginning and end of a function (see Table A–24 and Table A–25)
Symbol name	C_AUTO, C_STAT, C_MOE, C_MOS, C_MOU, C_TPDEF, C_EXT	DT_PTR DT_NON	(Any)	Symbol storage size in bits (see Table A–26)

In Table A–17, the structure tag name refers to any symbol name (including the special symbol .fake*n*/.tg*n*). The function name and array name refer to any symbol name. The section name includes .text, .data, and .bss.

Filenames

Each of the auxiliary table entries for a filename contains a 14-character file name in bytes 0–13. Bytes 14–17 are unused.

Table A–18. Filename Format for Auxiliary Table Entries

Byte Number	Type	Description
0–13	Character	File name
14–17	—	Unused

Sections

Table A–19 illustrates the format of auxiliary table entries for sections.

Table A–19. Section Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Long integer	Section length
4–5	Unsigned short integer	Number of relocation entries
6–7	Unsigned short integer	Number of line number entries
8–17	—	Not used (zero filled)

Tag Names

Table A–20 illustrates the format of auxiliary table entries for tag names.

Table A–20. Tag Name Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	—	Unused (zero filled)
4–7	Unsigned long integer	Size of structure, union, or enumeration
8–11	—	Unused (zero filled)
12–15	Long integer	Index of next entry beyond this structure, union, or enumeration
16–17	—	Unused (zero filled)

End of Structure

Table A–21 illustrates the format of auxiliary table entries for ends of structures.

Table A–21. End-of-Structure Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Long integer	Tag index
4–7	Unsigned long integer	Size of structure, union, or enumeration
8–17	—	Unused (zero filled)

Functions

Table A–22 illustrates the format of auxiliary table entries for functions.

Table A–22. Function Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Long integer	Tag index
4–7	Long integer	Size of function (in bytes)
8–11	Long integer	File pointer to line number
12–15	Long integer	Index of next entry beyond this function
16–17	—	Unused (zero filled)

Arrays

Table A–23 illustrates the format of auxiliary table entries for arrays.

Table A–23. Array Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Long integer	Tag index
4–7	Unsigned long integer	Size of array (in bits)
8–9	Unsigned short integer	First dimension
10–11	Unsigned short integer	Second dimension
12–13	Unsigned short integer	Third dimension
14–15	Unsigned short integer	Fourth dimension
16–17	—	Unused (zero filled)

End of Blocks and Functions

Table A–24 illustrates the format of auxiliary table entries for the ends of blocks and functions.

Table A–24. End-of-Blocks/Functions Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	—	Unused (zero filled)
4–5	Unsigned short integer	C source line number
6–17	—	Unused (zero filled)

Beginning of Blocks and Functions

Table A–25 illustrates the format of auxiliary table entries for the beginnings of blocks and functions.

Table A–25. Beginning-of-Blocks/Functions Format for Auxiliary Table

Byte Number	Type	Description
0–3	Unsigned long integer	Register save mask
4–5	Unsigned short integer	C source line number of block begin
6–7	Unsigned short integer	Number line entries for function
8–11	Unsigned long integer	Size of local frame for function
12–15	Long integer	Index of next entry past this block
16–17	—	Unused (zero filled)

Symbol Names

Table A–26 illustrates the format of auxiliary table entries for the symbol names.

Table A–26. Symbol Name Format for Auxiliary Table Entries

Byte Number	Type	Description
0–3	Long integer	Tag index
4–7	Unsigned long integer	Size of symbol (in bits)
8–17	—	Unused (zero filled)

Appendix B

The Linker Example Code

This appendix contains the full C code for the extended linker example in Chapter 14, *Linking PP and MP Files: An Extended Example*.

Topics

B.1 The MP Program, mp.c

```
/************************************************************************/
/* MP.C                                                                 */
/*                                                                      */
/*    MAIN() – This routine will calculate a table of function results.     */
/*             This routine will calculate the results, by initiating       */
/*             one function per PP. The MP will wait for the results,       */
/*             and will print them in a table.                              */
/*                                                                      */
/************************************************************************/
#include <stdio.h>
#include <mvp.h>

/************************************************************************/
/* Define PP control macros                                             */
/************************************************************************/
#define INTER_PPS      0x000F0000
#define RESET_PPS          asm(”\tcmnd 0x8000000F”)
#define START_PPS          asm(”\tcmnd 0x3000000F”)
#define PPSTART_VEC(pp)    *(int **)(0x010001b8 + ((pp) << 12))
#define TABLE_MAX          10

/************************************************************************/
/* Allocate space for the communication buffers in their own section. This  */
/* will help avoid cache errors.                                        */
/************************************************************************/
#pragma DATA_SECTION(comm_fact_num, ”.bypass”);
#pragma DATA_SECTION(comm_sum_num,  ”.bypass”);
#pragma DATA_SECTION(comm_fib_num,  ”.bypass”);
#pragma DATA_SECTION(comm_pow4_num, ”.bypass”);
#pragma DATA_SECTION(comm_fact,     ”.bypass”);
#pragma DATA_SECTION(comm_sum,      ”.bypass”);
#pragma DATA_SECTION(comm_fib,      ”.bypass”);
#pragma DATA_SECTION(comm_pow4,     ”.bypass”);

/************************************************************************/
/* Define the communication variables. They are defined shared so that      */
/* both MP and PP code can reference them.                              */
/************************************************************************/
shared int comm_fact_num, comm_sum_num, comm_fib_num, comm_pow4_num;

shared int comm_fact[TABLE_MAX], comm_sum [TABLE_MAX],
           comm_fib [TABLE_MAX], comm_pow4[TABLE_MAX];

/************************************************************************/
/* Entry points to the functions which will be run on the PPs.              */
/************************************************************************/
extern int ep_tsk_fact, ep_tsk_sum, ep_tsk_fib, ep_tsk_pow4;

/************************************************************************/
/* MAIN() – Driver routine for this example.                            */
/************************************************************************/
main()
{
    int i;

    /********************************************************************/
    /* Set PPs entry point vectors to function they will perform.           */
    /********************************************************************/
    PPSTART_VEC(0) = &ep_tsk_fib;
    PPSTART_VEC(1) = &ep_tsk_sum;
    PPSTART_VEC(2) = &ep_tsk_pow4;
    PPSTART_VEC(3) = &ep_tsk_fact;
```

```
    /*********************************************************************/
    /* Set up the argument vectors for each PP task.                     */
    /*********************************************************************/
    for (i=1; i < TABLE_MAX; ++i)
    {
        /*****************************************************************/
        /* We bypass cache, so the write goes directly to external mem   */
        /*****************************************************************/
        NOCACHE_INT(comm_fib[i])  = i;
        NOCACHE_INT(comm_sum[i])  = i;
        NOCACHE_INT(comm_pow4[i]) = i;
        NOCACHE_INT(comm_fact[i]) = i;
    }

    /*********************************************************************/
    /* Set the globals indicating how many arguments each PP will process.  */
    /*********************************************************************/
    NOCACHE_INT(comm_fib_num)  = TABLE_MAX-1;
    NOCACHE_INT(comm_sum_num)  = TABLE_MAX-1;
    NOCACHE_INT(comm_pow4_num) = TABLE_MAX-1;
    NOCACHE_INT(comm_fact_num) = TABLE_MAX-1;

    /*********************************************************************/
    /* Reset the PPs to get them in a known state.                       */
    /*********************************************************************/
    RESET_PPS;

    /*********************************************************************/
    /* Clear INTPEN flag. Each PP writes 1 into INTPEN, when done        */
    /*********************************************************************/
    INTPEN = ~0;

    /*********************************************************************/
    /* Ensure all PP's were halted by the reset, then start PP's         */
    /*********************************************************************/
    while ((PPERROR & INTER_PPS) != INTER_PPS);
    START_PPS;

    /*********************************************************************/
    /* Print table header.                                               */
    /*********************************************************************/
    printf("      i      fib      sum     pow4     fact     \n");
    printf("      _      ___      ___     ____     ____     \n");

    /*********************************************************************/
    /* Wait for all PP's to signal that they have completed.             */
    /*********************************************************************/
    while ((INTPEN & INTER_PPS) != INTER_PPS);

    /*********************************************************************/
    /* Read results generated by the PP's and print them out.            */
    /*********************************************************************/
    for (i=1; i < TABLE_MAX; ++i)
    {
        /*****************************************************************/
        /* Note that results are read directly from memory, because the PP's*/
        /* did not write them into cache.                                */
        /*****************************************************************/
        printf("%7d  %7d  %7d  %7d  %7d\n", i,
                NOCACHE_INT(comm_fib[i]),  NOCACHE_INT(comm_sum[i]),
                NOCACHE_INT(comm_pow4[i]), NOCACHE_INT(comm_fact[i]));
    }
}
```

B.2 The PP Factorial Programs, fact.c and fact_a.c

fact.c

```
/************************************************************************/
/* FACT.C                                                               */
/*                                                                      */
/*    MAIN() – This main routine will read a global array, compute      */
/*             the factorial of the arrays elements, and write the results*/
/*             back to the global array. It then will signal the MP that  */
/*             it has completed.                                        */
/*                                                                      */
/************************************************************************/
#include <mvp.h>

extern int           fact(int x);    /* Routine that calculates a factorial */
extern sharedpp void signal_done();  /* Routine that signals MP             */
extern shared int    comm_fact_num;  /* Global scalar indicating # of args  */
extern shared int    comm_fact[10];  /* Global Argument array               */
int                  onchip[10];     /* On–chip copy of argument array      */

main()
{
   int i;

   /************************************************************************/
   /* Block copy argument array on–chip, for faster access.                */
   /************************************************************************/
   memtrans(onchip, comm_fact, sizeof(comm_fact));

   for (i=1; i <= comm_fact_num; ++i) onchip[i] = fact(onchip[i]);

   /************************************************************************/
   /* Block copy argument array back to external memory.                   */
   /************************************************************************/
   memtrans(comm_fact, onchip, sizeof(comm_fact));

   signal_done();
}
```

fact_a.c

```
/************************************************************************/
/* FACT_A.C                                                             */
/*                                                                      */
/*    FACT() – This function will compute the factorial of its argument.  */
/*                                                                      */
/************************************************************************/

int fact(int x)
{
   int i=1, result=1;

   while (i <= x)  result *= i++;
   return result;
}
```

B.3 The PP Fibonacci Programs, fib.c and fib_a.c

fib.c

```
/*************************************************************************/
/* FIB.C                                                                 */
/*                                                                       */
/*    MAIN() – This main routine will read a global array, compute the   */
/*             ith position of a Fibonacci sequence, where i is defined  */
/*             by the value of the array elements, and write the results */
/*             back to the global array. It then will signal the MP that */
/*             it has completed.                                         */
/*                                                                       */
/*************************************************************************/
#include <mvp.h>

extern int           fib(int x);     /* Routine that calculates Fibonacci   */
extern sharedpp void signal_done();  /* Routine that signals MP             */
extern shared int    comm_fib_num;   /* Global scalar indicating # of args  */
extern shared int    comm_fib[10];   /* Global Argument array               */
int                  onchip[10];     /* On–chip copy of argument array      */

main()
{
   int i;

   /*************************************************************************/
   /* Block copy argument array on–chip, for faster access.                 */
   /*************************************************************************/
   memtrans(onchip, comm_fib, sizeof(comm_fib));

   for (i=1; i <= comm_fib_num; ++i) onchip[i] = fib(onchip[i]);

   /*************************************************************************/
   /* Block copy argument array back to external memory.                    */
   /*************************************************************************/
   memtrans(comm_fib, onchip, sizeof(comm_fib));

   signal_done();
}
```

fib_a.c

```
/*************************************************************************/
/* FIB_A.C                                                               */
/*                                                                       */
/*    FIB() – This function will return the ith position of the Fibonacci seq.  */
/*                                                                       */
/*************************************************************************/

int fib(int x)
{
   int i, last = 1, fibi = 1;

   for (i=2; i < x; ++i) { int tmp = fibi + last; last = fibi; fibi = tmp; }
   return fibi;
}
```

B.4 The PP Power of Four Program, pow4.c

```
/*************************************************************************/
/* POW4.C                                                                */
/*                                                                       */
/*    MAIN() – This main routine will read a global variable, call the   */
/*             PP runtime–support function POW() to compute the variable */
/*             raised to the 4th power, and write the result back to the */
/*             global variable. It then will signal the MP that it has   */
/*             completed.                                                */
/*                                                                       */
/*************************************************************************/
#include <math.h>
#include <mvp.h>

extern sharedpp void signal_done();  /* Routine that signals MP            */
extern shared int    comm_pow4_num;  /* Global scalar indicating # of args */
extern shared int    comm_pow4[10];  /* Global Argument array              */
int                  onchip[10];     /* On–chip copy of argument array     */

main()
{
   int i;

   /*********************************************************************/
   /* Block copy argument array on–chip, for faster access.             */
   /*********************************************************************/
   memtrans(onchip, comm_pow4, sizeof(comm_pow4));

   for (i=1; i <= comm_pow4_num; ++i) onchip[i] = pow(onchip[i], 4.0);

   /*********************************************************************/
   /* Block copy argument array back to external memory.                */
   /*********************************************************************/
   memtrans(comm_pow4, onchip, sizeof(comm_pow4));

   signal_done();
}
```

B.5 The PP Summation Programs, sum.c and sum_a.c

sum.c

```
/*************************************************************************/
/* SUM.C                                                                 */
/*                                                                       */
/*    MAIN() – This main routine will read a global array, compute       */
/*             the sum of integers from 1 to the value of the the arrays */
/*             elements, and write the sums back to the global array.    */
/*             It then will signal the MP that it has completed.         */
/*                                                                       */
/*************************************************************************/
#include <mvp.h>

extern int           sum(int x);     /* Routine that calculates a summation */
extern sharedpp void signal_done();  /* Routine that signals MP             */
extern shared int    comm_sum_num;   /* Global scalar indicating # of args  */
extern shared int    comm_sum[10];   /* Global Argument array               */
int                  onchip[10];     /* On–chip copy of argument array      */

main()
{
   int i;

   /*********************************************************************/
   /* Block copy argument array on–chip, for faster access.             */
   /*********************************************************************/
   memtrans(onchip, comm_sum, sizeof(comm_sum));

   for (i=1; i <= comm_sum_num; ++i) onchip[i] = sum(onchip[i]);

   /*********************************************************************/
   /* Block copy argument array back to external memory.                */
   /*********************************************************************/
   memtrans(comm_sum, onchip, sizeof(comm_sum));

   signal_done();
}
```

sum_a.c

```
/*************************************************************************/
/* SUM_A.C                                                               */
/*                                                                       */
/*    SUM() – This function will compute the summation of its argument.  */
/*                                                                       */
/*************************************************************************/

int sum(int x)
{
   int i=1, result=0;

   while (i <= x)  result += i++;
   return result;
}
```

B.6 The Linker Command Files

mplnk.cmd

```
-c
-x
-heap  0x2000
-stack 0x2000
-l mp_rts.lib

MEMORY
{
    EXTMEM    : o=0x02000000   l = 0x80000
}

SECTIONS
{
    .text    :  > EXTMEM
    .ptext   :  > EXTMEM
    .bss     :  > EXTMEM
    .const   :  > EXTMEM
    .switch  :  > EXTMEM
    .sysmem  :  > EXTMEM
    .stack   :  > EXTMEM
    .cinit   :  > EXTMEM
    .pcinit  :  > EXTMEM
}
```

pplnk.cmd

```
-pc
-x
-pheap  0x800
-pstack 0x580
-l pp_rts.lib

MEMORY
{
    DRAM01    : o=0x00000004   l = 0x00ffc
    DRAM2     : o=0x00008000   l = 0x00800
    PRAM0     : o=0x01000200   l = 0x00600
    EXTMEM    : o=0x02000000   l = 0x80000
}

SECTIONS
{
    .ptext   :  > EXTMEM
    .const   :  > EXTMEM
    .switch  :  > EXTMEM
    .pcinit  :  > EXTMEM
    .bss     :  > EXTMEM
    .pbss    :  (PASS) > DRAM01
    .psysmem :  (PASS) > DRAM2
    .pstack  :  (PASS) > PRAM0
}
```

example.cmd

```
–u _exit
–l mp_cio.lib

SECTIONS
{
    .cio     :  > EXTMEM
    .bypass  : { *(.bypass) . = align(64); } align(64) > EXTMEM
}
```

Appendix C

Glossary

A

alignment: A process in which the linker places an output section or element at an address that falls on an n-byte boundary, where n is a power of 2.

allocation: A process in which the linker calculates the final memory addresses of relocatable output sections.

allocation node: The processor node into which an internode message is allocated.

archive library: A collection of individual files that have been grouped into a single file.

archiver: A software program that allows you to collect several individual files into an archive library.

assembler: A software utility that creates a machine-language program from a source file. There are two assemblers associated with the MVP: a mnemonic-based RISC-type assembler for the MP and an algebraic assembler for the PP.

assembly-time constant: A symbol that is assigned a constant value with the .set or .equ directive. Such values are assigned at assembly time.

autoinitialization: The process of initializing global C variables (in the .cinit section) before beginning program execution.

auxiliary entry: The extra entry that a symbol may have in the symbol table. It contains additional information about the symbol (whether the symbol is a filename, a section name, a function name, etc.).

B

binding: Associating or linking together two complementary software objects.

big endian: An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is specific to hardware and is determined at reset. See also *little endian*

C

C compiler: A program that translates C source statements into assembly language source statements or object code.

cache: A fast memory into which frequently used data or instructions from slower memory are copied for fast access. Fast access is facilitated by the cache's high speed and its on-chip proximity to the CPU.

cache clean: An MP instruction that updates external memory by writing modified (dirty) data-cache subblocks back to memory, thus resetting that subblock's dirty bit to 0.

cache flush: An MP instruction that updates external memory by writing modified (dirty) data-cache subblocks back to memory, thus resetting that subblock's present and dirty bits to 0.

clean: See *cache clean*

COFF: *Common object file format*. An object file format that promotes modular programming by supporting sections.

COFF magic number: A COFF file header entry that identifies an object file as a module that can be executed by the TMS320C8x. See also *target ID*

common object file format: See *COFF*

cross-reference listing: An output file created by the assembler that lists the symbols that were defined, the line on which they were defined, any lines that referenced them, and their final values.

D

data cache: The MP's two SRAM banks that hold cached data needed by the MP. Data RAMs for the PPs are not cached.

debugger: A window-oriented software interface that helps you to debug MVP programs running on an MVP emulator or simulator.

directives: Special-purpose commands that control the actions and functions of a software tool (as opposed to assembly language instructions, which control the actions of a device).

dirty flag: A storage bit associated with each subblock of MP data-cache memory that indicates whether the subblock contains modified data that needs to be written back to main memory. See also *cache flush*, *cache clean*

doubleword: A 64-bit value.

E

emulator: A debugging tool that is external to the target system and that provides direct control over the MVP that is in the target system.

entry point: The starting execution point for a piece of code in target memory.

executable module: An object file that has been linked and can be executed in an MVP system.

executive: The portion of a multitasking software system that is responsible for executing application tasks, providing communications among tasks, and managing shared resources.

external address: See *off-chip address*

external symbol: A symbol that is used in the current program module but that may be defined in a different program module.

F

flush: See *cache flush*

G

global symbol: A symbol that is either defined in the current module and accessible from other modules or is accessed in the current module but may be defined in another.

H

halfword: A 16-bit value.

hex conversion utility: A tool that translates COFF object files into one of several standard ASCII hexadecimal formats suitable for loading into an EPROM programmer.

I

incremental linking: The further linking of files that have already been linked. When linking files in this manner, be sure to include the –r option so that relocation information will be included, and do not use the –s option, which strips symbolic information.

initialized section: A COFF section that contains executable code or initialized data. An initialized section can be built using the .data, .ptext, .text, or .sect directive.

input section: A section from an object file that will be linked into an executable module.

internal address: See *on-chip address*

interprocessor command: A message sent via the crossbar to the other on-chip processors.

L

linker: A software tool that combines object files to form an object module that can be allocated into system memory and executed by the device.

listing file: An output file created by the assembler that lists source statements, their line numbers, and their effects on the SPC.

little endian: An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher numbered addresses. Endian ordering is specific to hardware and is determined at reset. See also *big endian*

loader: A device that loads an executable module into system memory.

LSB: *Least significant bit.* The bit having the smallest effect on the value of a binary numeral, usually the rightmost bit. The MVP numbers the bits in a word from 0 to 31, where bit 0 is the LSB.

M

macro: A user-defined routine that executes as an instruction.

macro call: The process of invoking a macro.

macro definition: A block of source statements that define the name and the code that make up a macro. See also *model statement*

macro expansion: The source statements that are substituted for the macro call and are subsequently assembled.

macro library: An archive library composed of macros. Each file in the library must contain one macro; its name must be the same as the macro name it defines, and it must have an extension of .asm.

magic number: See *COFF magic number*

map file: An output file, created by the linker, that shows the memory configuration, section composition, and section allocation, as well as symbols and the addresses at which they were defined.

master processor: See *MP*

member: The elements or variables of a structure, union, or enumeration.

memory map: A map of target system memory space that is partitioned into functional blocks.

mnemonic: An instruction name that the assembler translates into machine code.

model statement: Instructions or assembler directives in a macro definition that are assembled each time a macro is invoked.

MP: *Master processor*. A general-purpose RISC processor that coordinates the activity of the other processors on the MVP. The MP includes an IEEE-754 floating-point hardware unit.

mpcl: A shell utility that invokes the MVP master processor compiler, assembler, and linker to create an executable object file version of your MP program.

MSB: *Most significant bit*. The bit having the greatest effect on the value of a binary numeral. It is the leftmost bit. The MVP numbers the bits in a word from 0 to 31, where bit 31 is the MSB.

multimedia video processor: See *MVP*

MVP: *Multimedia video processor*. A single-chip multiprocessor device that accelerates applications such as video compression and decompression, image processing, and graphics. The multimedia video processor contains a master processor and from one to eight parallel processors, depending on the device version. For example, the TMS320C80 device contains four PPs.

MVP multitasking executive: See *executive*

N

named section: An initialized section that is defined with a .sect directive or an uninitialized section that is defined with a .usect directive.

O

object file: A file that has been assembled or linked and contains machine-language object code.

object format converter: A program that converts COFF object files into Intel-format, Tektronix-format, TI-tagged-format, or Motorola-S-format object files.

object library: An archive library comprised of individual object files.

off-chip address: An address external to the MVP chip. Addresses from 0x0200 0000 to 0xFFFF FFFF are off-chip addresses. See also *on-chip address*

on-chip address: An address internal to the MVP chip. Addresses from 0x0000 0000 to 0x1FFF FFFF are on-chip addresses. See also *off-chip address*

optional header: A portion of a COFF object file that the linker uses to perform relocation at download time.

output module: A linked, executable object file that can be downloaded and executed on a target system.

P

parallel processor: See *PP*

parameter RAM: A general-purpose 2K-byte RAM that is associated with a specific processor, part of which is dedicated to packet transfer information and the processor interrupt vectors.

partial linking: See *incremental linking*

PP: *Parallel processor*. The MVP's advanced digital signal processor that is used for video compression/ decompression (P × 64 or MPEG), still-image compression/ decompression (JPEG), 2-D and 3-D graphic functions such as line draw, trapezoid fill, antialiasing, and a variety of high-speed integer operations on image data. An MVP single-chip multiprocessor device may contain from one to eight PPs, depending on the device version.

ppcl: A shell utility that invokes the MVP's PP compiler, assembler, and linker to create an executable object file version of your PP program.

R

RAM model: An autoinitialization model used by the linker when linking C code, which allows variables to be initialized at load time instead of runtime. It is used when you invoke the linker with the –cr option.

relocation: A process in which the linker adjusts all the references to a symbol when the symbol's address changes. Also refers to the linker's process of placing each output section in memory.

ROM model: An autoinitialization model used by the linker in which the linker loads the .cinit section of the data tables into memory and variables are initialized at runtime. It is used when you invoke the linker with the – c option.

S

section: A relocatable block of code or data that will ultimately occupy contiguous space in the memory map.

section header: A portion of a COFF object file that contains information about a section in the file. Each section has its own header; the header points to the section's starting address and contains the section's size, etc.

section program counter: See *SPC*

sign-extend: To fill the unused MSBs of a value (register contents) with the value's sign bit.

SPC: S*ection program counter*. An element of the assembler that keeps track of the current location within a section. Each section has its own SPC.

stack pointer: A special-purpose 32-bit register that contains (points to) the address of the top of the system stack.

static variable: Variable whose scope is confined to a function or a program. The values of static variables are not discarded when the function or program is exited; their previous value is resumed when the function or program is reentered.

storage class: Any entry in the symbol table that indicates how a symbol should be accessed.

string table: A table that stores symbol names that are longer than eight characters (symbol names of eight characters or longer cannot be stored in the symbol table; instead, they are stored in the string table).

substitution symbol table: A table that is maintained by the assembler during the assembler execution that keeps track of the text to be associated with given symbols.

symbol: A string of alphanumeric characters that represents an address or a value.

symbol table: A portion of a COFF object file that contains information about the symbols that are defined and used by the file.

symbolic debugging: The ability of a software tool to retain symbolic information so that it can be used by a debugging tool such as a simulator or an emulator.

T

tag: 1) An optional type name that can be assigned to a structure, union, or enumeration. 2) A register holding the address of the cache block.

target ID: An ID associated with the COFF magic number that identifies the processor on which this file may be executed.

target memory: Physical memory in an MVP-based system into which executable object code is loaded.

TC: *Transfer controller*. The MVP's on-chip DMA controller for servicing the cache and for transferring one-, two-, and three-dimensional data blocks between each processor on the MVP and its external memory.

transfer controller: See *TC*

trap: An exceptional condition caused by the currently executing instruction that forces a program to be interrupted before execution of the next instruction begins. After the processor has serviced the trap, it typically resumes execution of the interrupted program at the instruction that immediately follows the instruction that caused the trap.

U

unconfigured memory: Memory that is not defined as part of the memory map and cannot be loaded with code or data.

uninitialized section: A COFF section that reserves space in the memory map but that has no actual contents. These sections are built with the .bss, .reg, and .usect directives.

union: A data structure similar to a C union that allows you to access the same storage area using different instruction data types. The assembler assigns symbolic offsets to union members to be allocated in the same memory space, creating a template that can be used as often as necessary.

unsigned: A value that is treated as a positive number, regardless of its actual sign.

V

VC: *Video controller*. The portion of the MVP responsible for the video interface.

video controller: See *VC*

W

well-defined expression: An expression that contains only symbols or assembly-time constants that have been defined before they appear in the expression.

word: A sequence of 32 adjacent bits that constitutes a register or memory value. The PP supports 32-bit words. The MP also supports doublewords of 64 bits for loads and stores.

X

xba: The assembler keyword for a PP-relative base address in local RAM, either data or parameter RAM. After memory allocation is performed by the linker, xba is changed to either dba or pba, depending on where space is allocated by the linker.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations and notices. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Resale of TI's products or services with *statements different from or beyond the parameters* stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products. www.ti.com/sc/docs/stdterms.htm

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265