MECCUM/D

REV 1

JANUARY 1997

MOTOROLA

EMBEDDED C COMPILER (MECC) Version 2.0 USER'S MANUAL

Important Notice to Users

While every effort has been made to ensure the accuracy of all information in this document, Motorola assumes no liability to any party for any loss or damage caused by errors or omissions or by statements of any kind in this document, its updates, supplements, or special editions, whether such errors are omissions or statements resulting from negligence, accident, or any other cause. Motorola further assumes no liability arising out of the application or use of any information, product, or system described herein; nor any liability for incidental or consequential damages arising from the use of this document. Motorola disclaims all warranties regarding the information contained herein, whether expressed, implied, or statutory, *including implied warranties of merchantability or fitness for a particular purpose.* Motorola makes no representation that the interconnection of products in the manner described herein will not infringe on existing or future patent rights, nor do the descriptions contained herein imply the granting or license to make, use or sell equipment constructed in accordance with this description.

Trademarks

This document includes these trademarks:

Motorola and the Motorola logo are registered trademarks of Motorola, Inc.
AIX, IBM, and PowerPC are trademarks of International Business Machines Corporation.
SPARC is a trademark of SPARC international, Inc.
Sun and SunOS are trademarks of Sun Microsystems, Inc.
UNIX is a trademark of Novell, Inc., in the United States and other countries, licensed exclusively through X/Open Company, Ltd.
X Window System is a trademark of Massachusetts Institute of Technology.

Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

CONTENTS

CHAPTER 1 INTRODUCTION

CHAPTER 2 USING THE COMPILER

CHAPTER 2 USING THE COMPILER (Continued)

CHAPTER 3 CONTROL VARIABLES

CHAPTER 3 CONTROL VARIABLES (Continued)

CHAPTER 4 COMPILER OPTIMIZATIONS

CHAPTER 4 COMPILER OPTIMIZATIONS (Continued)

CHAPTER 5 EMBEDDED APPLICATION BINARY INTERFACE

APPENDIX B LANGUAGE DIALECTS

APPENDIX C C RUN-TIME LIBRARIES

FIGURES

TABLES

CHAPTER 1
INTRODUCTION

The Motorola Embedded C Compiler (MECC) is an optimizing compiler for C-language source files. The compiler supports full ANSI C, as well as many extensions to the C language. The compiler generates code targeted for various PowerPC™ implementations.

The compiler is one part of the MPCCOMPKG-series Motorola Embedded PowerPC C Compiler Package. Other parts of the compiler package are the Motorola Embedded Assembler (MEAS), the Motorola Link Editor (MELD), the Motorola Archiver (MAR), and the Motorola S-Record Generator (MSREC).

The compiler accepts three C dialects: Kernighan and Ritchie (K&R) C, ANSI C, and a less strict form of ANSI C (relaxed C).

NOTE

If you use the Motorola Embedded Project (MEPROJ) to develop your code, the MEPROJ graphical user interface gives you direct access to the compiler.

1.1 MECC OVERVIEW

Commonly, you want to generate an executable program from a set of source files. You may submit all the files to the compiler via one command line (this is the default arrangement). The compiler:

- Analyzes the C-language files.
- Translates the files into optimized PowerPC assembler files. (Chapter 4 explains the kinds of optimization the compiler can perform.)
- Calls the assembler, which produces relocatable object files.
- Calls the link editor, which binds the relocatable object files into an executable object file.

During program development, however, you only need to compile specific files that have changed. Accordingly, you may submit individual source files to the compiler, which produces individual relocatable object files. You can use a final MECC invocation to produce an executable object file from such multiple relocatable object files.

Other ways to use the compiler include:

- Creating assembler files from C source code, without producing relocatable object files.
- Submitting assembler files, to produce relocatable object files.
- Submitting relocatable object files, to produce an executable object file.

When you invoke the compiler, it checks the type of each input file, looks for any command-line options, then carries out compilation accordingly.

Once the compiler has compiled the files, it calls the assembler to produce object code, then calls the link editor to produce an executable object file. Control options give you extensive control over these actions.

1.2 SYSTEM REQUIREMENTS

The compiler runs on specific computers and operating systems. For software installation instructions, see the appropriate Software Release Guide for your MPCCOMPKG-series Motorola Embedded PowerPC C Compiler Package. This guide also includes version information for the operating system software.

1.3 MECC USERS

To get the most benefit from this manual, you should understand embedded MCU applications and you should have C-language programming experience.

NOTE

If you are a beginning C programmer, Motorola recommends *The C Programming Language*, by Kernighan and Ritchie.

1.4 MANUAL CONVENTIONS

Table 1-1 lists the syntax and typographical conventions of this manual.

Table 1-1. Manual Conventions

Symbol, Typeface	Significance
Courier bold	Filenames, variables, commands, control variables, and examples.
Courier bold italic	Syntax indicator to be replaced by an actual value.
[ ]	Indicates optional values.
( )	Groups or delineates values.
*	Indicates a value that occurs zero or more times.
|	Indicates a choice between two values.

The remaining sections and appendixes of this manual cover these topics:

- Section 2: how to invoke the compiler, including command-line options.
- Section 3: how to use control variables, both in the command line and in pragma directives.
- Section 4: optimizations that the compiler makes possible.
- Section 5: PowerPC embedded application binary interface (EABI) compatibility.
- Appendix A: MECC error messages.
- Appendix B: language dialects.
- Appendix C: C run-time libraries.

CHAPTER 2
USING THE COMPILER

This section explains how to configure the environment variables and how to invoke the compiler. In addition, this section explains the many command-line control options.

Before using the compiler, you must install MECC and other development tools software, per instructions of the compiler-package Software Release Guide.

2.1 CONFIGURING ENVIRONMENT VARIABLES

After you have installed the software, you must set the environment variables **TMPDIR**, **PPC_BIN**, and **PATH**. **TMPDIR**, the simplest of these variables, merely contains the name of the directory in which the compiler stores temporary files. Paragraphs 2.1.1 through 2.1.3 explain the other environment variables.

You also should be aware of the special preprocessor symbol __**MOTO**__, which you may use in preprocessor conditional structures. Paragraph 2.3.10 explains this symbol.

2.1.1 The PPC_BIN Environment Variable

Set the **PPC_BIN** environment variable to the directory that contains the compiler executables. For example, if you install the compiler in the directory **c:\ppc**, make sure that **PPC_BIN** is set to that value. This makes sure that the compiler can find its executable components.

2.1.2 The PATH Environment Variable

The value of the **PATH** environment variable is a list of directories and subdirectories to be searched for MECC executables. Accordingly, the **PATH** value must include the directory that contains these executables.

For example, if you install the compiler in the directory **c:\ppc**, your **PATH** environment variable might be

```
.;c:\bin;c:\ppc
```

2.2 INVOKING THE COMPILER

To invoke the compiler, enter the compiler command line at your command prompt. This command line consists of the command **mecc**, followed by control options and filenames. The control options may include specifications for control variables.

The syntax of the command line is:

```
mecc [-opt1 -opt2... -optn] file1[ file2... filen]
```

where **-opt1 -opt2...-optn** represent control options, and **file1, file2... filen** represent source files. Note that spaces must separate options and filenames in the command line.

For example,

```
mecc -c test1.c test2.c test3.c
```

invokes the compiler for files **test1.c**, **test2.c**, and **test3.c**, producing an object file for each file. The **-c** control option tells the compiler to produce object files only. Paragraph 2.3 explains more about control options.

All the example filenames ended in **.c**, indicating that they are C source files. Filename extensions identify file types. The standard extensions are:

- **.c** or **.C** — for C-language source files
- **.i** — for preprocessed source files (which the compiler produces if you use the **-P** control option)
- **.s** or **.S** — for assembly-language source files
- **.o** — for relocatable object files
- **.a** — for library (archive) files

The default arrangement is that the compiler processes all the files you specify in the command line, outputting a single executable object file. (If the **mecc** command line specifies files that have no extensions, or have extensions not in the list above, the compiler passes the files to the link editor.)

Accordingly, the command line:

```
mecc test1.c test2.c test3.c
```

compiles the three files, assembles the files, then calls the link editor to link the files. The compiler assigns the default name **a.out** to the executable object file.

The **-o** option lets you specify the name of the executable object file:

```
mecc -o test test1.c test2.c test3.c
```

In this example, the compiler compiles the same three files, assembles the files, and also calls the link editor. The link editor outputs an executable object file, but the **-o** option directs the link editor to name the file **test** instead of the default name **a.out**. (If the compiler not only compiles, but also calls the assembler and link editor, it deletes any object files produced during compilation.)

A fourth example includes C (**.c**) files, an assembly-language (**.s**) file, and an object (**.o**) file:

```
mecc -O3 -o test test1.c test2.c test3.s test4.o test5
```

The compiler:

1. Compiles the two C files, producing object files **test1.o** and **test2.o**. During compilation the compiler carries out the second level of optimization (per the **-O3** option). The C compiler does not compile files **test3.s**, **test4.o**, or **test5**.
2. Calls the assembler to assemble file **test3.s**.
3. Calls the link editor, which produces an executable object file from all the object files produced during steps 1 and 2, plus file **test5**. Per the **-o** option, the link editor names this executable object file **test**.

2.3 CONTROL OPTIONS

The compiler command line examples of paragraph 2.2 show how control options modify the steps of compilation, assembly, and linking. This paragraph explains the basic control options.

Note that some control options conflict. Should you include conflicting control options in the command line, a warning message indicates that the compiler ignored one or more of the options.

Table 2-1 lists the control options. Paragraphs 2.3.1 through 2.3.23 give additional detail about these options.

Table 2-1. MECC Control Options

Option	Effect
-c	Does not link files.
-g	Includes debugging information.
-l	Searches the specified library.
-o	Names the executable object file.
-v	Prints process names and arguments.
-w	Suppresses warning messages.
-A	Assigns control-variable value; permits pragma overrides.
-AA	Assigns control-variable value; does not permit pragma overrides.
-C	Retains comments in preprocessor output.
-D	Defines preprocessor symbols.
-E	Limits MECC to preprocessing.
-H	Lists preprocessing pathnames.
-I	Adds a search path for include files.
-K	Compiles Kernighan & Ritchie (K&R) C.
-L	Changes the search order for libraries.
-M	Lists dependencies.
-O	Activates optimization.
-P	Limits MECC to preprocessing.
-S	Compiles only: produces, but does not assemble, an assembly-language file.
-U	Undefines preprocessor symbols.
-V	Prints the version number of the compiler.
-W	Hands listed arguments to the specified compiler pass.
@<pathname>	Process options in specified file as if they were part of the command line.

2.3.1 Suppress Linking Option (-c)

The **-c** control option tells the compiler to *not* call the link editor. If you include this option in the command line, the compiler compiles and assembles **.c** or **.i** files, then calls the MEAS to assemble any **.s** files. But as the compiler does not call the link editor, the **.o** files are the final output.

2.3.2 Include Debugging Option (-g)

The **-g** control option tells the compiler to include symbolic debugging information (source annotations, symbols, and line numbers) in output files, for use with a symbolic debugger.

If the **mecc** command line includes **.s** files, for which the compiler calls the assembler, the output **.o** files include the debugging information. For **.c** or **.i** files of the **mecc** command line, the output **.s** files include the debugging information. However, assembly of such output **.s** files would produce output **.o** files that do not include debugging information.

2.3.3 Search Library Option (-l)

The **-l** option directs the link editor to search a specified library for object files that can resolve references during linking. For example, options **-lchart**, **-lmaps**, and **-lb** direct the link editor to search libraries **libchart.a**, **libmaps.a**, and **libb.a**, respectively. If you use this option several times in the command line, the link editor carries out the searches in the order of your command-line use.

2.3.4 Name Executable Option (-o)

The **-o** option assigns a specified filename to any of the output files. Note that the default executable filename is **a.out**. If your command line includes the option **-o outfile**, the link editor assigns the name **outfile** to the executable file.

2.3.5 Print Process Option (-v)

The **-v** option tells the compiler to print names and arguments as it invokes each compiler sub-process. (This information is helpful for diagnostics.)

2.3.6 Suppress Warnings Option (-w)

The **-w** option suppresses MECC warning messages.

2.3.7 Assign Value Option (-A)

The **-A** control option assigns a value to one or more control variables, but permits pragma directives in the source code to override the value assignments. (Section 3 gives the complete syntax for such value assignments.)

2.3.8 Assign Value Option (-AA)

The **-AA** control option assigns a value to one or more control variables, but does *not* permit pragma directives in the source code to override the value assignments. (Section 3 explains the complete syntax for such value assignments.)

2.3.9 Retain Preprocessor Comments Option (-C)

The **-C** control option directs the compiler to retain comments in the preprocessor output.

2.3.10 Define Preprocessor Symbol Option (-D)

The **-D** control option defines symbols for source files passed through the preprocessor. For example, the options **-Drock**, **-Dscissors**, and **-Dpaper** define the preprocessor symbols **rock**, **scissors**, and **paper**. This control option is the command-line counterpart to a **#define** directive in source code.

To assign a value at the same time you define a symbol, include the equals sign (=) and the value. For example, the option **-Dversion=4** defines the preprocessor symbol **version** and gives the symbol the value *4*.

NOTE

In case of conflict between **-D** control options, the *first* **-D** option of the command line determines the compiler's behavior. (For other embedded-tools command-line options, the *last* determines the behavior of the tool.) For example, if an **mecc** command line includes the options **... -Dalpha=100 ... -Dalpha=300 ... -Dalpha=800...**, the compiler gives the value 100 to symbol **alpha**.

A common use for such symbols is delimiting code that the preprocessor expands conditionally. For example, assume that the command line includes the option **-Dset_zero**, and that the source code includes the line **#ifdef set_zero**. The **-D** option defines the symbol **set_zero**, so the condition of the **#ifdef** directive is true. Consequently, the preprocessor includes the code that immediately follows the **#ifdef** directive.

NOTE

To undefine a symbol, use the **-U** control option. However, the **-D** control option does not override **#define** or **#undef** directives in source code.

The compiler automatically defines the preprocessor symbol **__MOTO__**, which you may use in source-code conditional structures. If you use the MECC to compile, the code expression **#ifdef __MOTO__** evaluates to 1 or TRUE. If you use a different compiler, the expression **#ifdef __MOTO__** evaluates to 0 or FALSE.

This simple code extract prints one comment if **__MOTO__** is defined, and a different comment if **__MOTO__** is not defined:

```
.
.
#ifdef __MOTO__
    printf("Compiled by MECC");
#else
   printf("Not Compiled by MECC");
#endif
```

2.3.11 Preprocess-Only Option (-E)

The **-E** control option directs the compiler to carry out preprocessing, directing the stream of output to **stdout**, then stop. The compiler does not complete compilation, does not call the assembler, and does not call the link editor. The preprocessing results do not include comments (unless the command line also has the **-C** option). This control option is similar to the **-P** option, which produces a **.i** file.

2.3.12 List Preprocessing-Pathname Option (-H)

The **-H** control option directs the compiler to pass source files through the preprocessor, listing pathnames of any included files to **stdout**, then stop. The I/O stream **stdout** receives the pathname list instead of the normal preprocessor results. The compiler does not complete compilation, does not call the assembler, and does not call the link editor.

2.3.13 Include-File Search Option (-I)

The **-I** control option specifies the search order for include files. The default arrangement is that the system searches through the directories of the standard list. (Installation of the compiler includes definition of this list, which includes the directory **$PPC_BIN/include**.)

Each **-I** option adds another directory in which the compiler searches for include files. The search proceeds according to the format of the filename **file** specified in the **#include** statement:

- If filename **file** is an absolute pathname, the system searches only that path.
- If filename **file** is not an absolute pathname *but is in quotes* in the **#include** statement, and if control variable **inclpath** has the value **absolute**, the system searches directories in this order:
  1. The directory that contains the primary C source file.
  2. Directories specified by any **-I** options, in their order in the command line.
  3. Directories in the standard list.
- If filename **file** is not an absolute pathname *but is in quotes* in the **#include** statement, and if control variable **inclpath** has the value **relative**, the system searches directories in this order:
  1. The directory of the file that contains the **#include** statement.
  2. Directories specified by any **-I** options, in their order in the command line.
  3. Directories in the standard list.
- If filename **file** is not an absolute pathname *but is in angles* (< >) in the **#include** statement, the system searches directories in this order:
  1. Directories specified by any **-I** options, in their order in the command line.
  2. Directories in the standard list.

2.3.14 Compile K&R Option (-K)

The **-K** control option directs the compiler to compile **.c** source files according to the Kernighan and Ritchie (K&R) dialect of C.

2.3.15 Library Search Option (-L)

The **-L** control option specifies the search order for library files. For example, the option **-L dir** tells the link editor to search for library files in directory **dir**. The compiler always tells the link editor to search in directory **$PPC_BIN/lib**. If you specify searches of any additional directories, the compiler tells the link editor to search in those directories first.

2.3.16 List Dependencies Option (-M)

The **-M** control option directs the compiler to pass source files through the preprocessor, listing dependency lines to **stdout**, then stop. The compiler determines the dependency lines according to the usage of **#include** statements (including nested ones). The I/O stream **stdout** receives the dependency-line list instead of the normal preprocessor results. The compiler does not complete compilation, does not call the assembler, and does not call the link editor.

2.3.17 Optimization Level Option (-O)

The **-O** control option assigns the level of optimization the compiler applies to source code. The general meaning of **-O** values is:

- **-O1** means do *no* optimization. (Not including the **-O** option in your command line gives this level of optimization.)
- **-O2** is reserved for future use. (At present, it defaults to **-O3**.)
- **-O3** means do local optimization, *register allocation*, *intraprocedural global optimization*, and *scheduling*. Including the **-O** option in your command line, but without a numeric value, gives this level of optimization.)
- **-O4** means do local optimization, register allocation, intraprocedural global optimization, scheduling, and *additional aggressive optimizations*.

Section 4 explains each type of optimization.

2.3.18 Preprocess-Only Option (-P)

The **-P** control option directs the compiler to carry out preprocessing, producing a **.i** file, then stop. The compiler does not call the assembler, and does not call the link editor. The preprocessing results do not include comments (unless the command line also has the **-C** option). This control option is similar to the **-E** option, which directs the I/O stream to **stdout**, instead of producing a **.i** file.

2.3.19 Compile-Only Option (-S)

The **-S** control option directs the compiler to carry out compilation, then stop. The compiler does not call the assembler or the link editor. The output of compilation is one or more **.s** files, which can be assembled at a later time.

2.3.20 Undefine Preprocessor Symbol Option (-U)

The **-U** control option undefines preprocessor symbols created via the **-D** option. For example, if the option **-Dset_zero** defines the symbol **set_zero**, the option **-Uset_zero** removes the symbol. The **-U** option may appear before or after the corresponding **-D** option.

NOTE

The **-U** control option does not override **#define** or **#undef** directives in source code.

2.3.21 Print Version Option (-V)

The **-V** option tells the compiler to print the version number of the compiler.

2.3.22 Pass Arguments Option (-W)

The **-W** control option tells the compiler to hand off the specified arguments to a particular pass of the compiler. For example, the option **-Wi,avg** says to hand off argument *avg* to the interprocedural analyzer pass of the compiler. The option **-Wa avg,med,mean** says to hand off arguments *avg*, *med*, and *mean* to the assembler pass.

The pass identifiers are:

- **p** pre-processor pass
- **f** front-end pass
- **i** interprocedural analysis pass
- **b** optimizer and code generator pass
- **a** assembler (MEAS) pass
- **l** link editor (MELD) pass

The syntax for specifying pass arguments is

`-Wx,arg`

where ***x*** represents the pass identifier and ***arg*** represents a non-space separated argument. For example,

```
-Wa,-DVERSION=10
```

tells the compiler to pass the argument **`-DVERSION=10`** to the assembly pass of compilation. (**`-DVERSION=10`** will be the first argument the assembler receives.)

To pass multiple arguments, specify **`-W`** multiple times, as in:

```
-Wa,-DVERSION=10 -Wa,-l -Wa,some_file.list
```

which tells the compiler to pass arguments **`-DVERSION=10`**, **`-l`**, and **`some_file.list`** to the assembly pass of compilation.

To confirm that the appropriate arguments reach the pass you intend, use the **`-v`** (print process option) option.

2.3.23 Command Line File Option (@)

The `@` control option tells the compiler to use the contents of a specified file as if the contents were part of the command line. For example, the option **`@cmd_opts`** tells the compiler to read file **`cmd_opts`**, considering the contents of the file to be part of the command line. (The specified file can include multiple lines of text.)

CHAPTER 3
CONTROL VARIABLES

Section 2 explained the various control options for the compiler command line. This section explains how to use control variables for finer control of compilation.

You can assign values to many of the control variables via pragma statements in the source code itself. The effect and positioning of a pragma statement in the source code depends on the particular control variable.

3.1 CONTROL VARIABLE BASICS

Each control variable controls an aspect of the compiler's behavior. To determine that behavior, you assign a specific value to the control variable or let the control variable have one of two possible default values. Table 3-1 (in paragraph 3.4) lists specifics for all the control variables.

When you invoke the compiler, the compiler assigns values to all control variables, in one of three ways:

1. If you include a value assignment in the command line, the compiler gives that value to the control variable. (Use the **`-A`** or **`-AA`** control option to assign a value.)
2. If you do not mention the control variable in the command line, the compiler gives the control variable its first default value.
3. If you mention the control variable in the command line, but do not assign a value, the compiler gives the control variable its second default value. (Use the **`-A`** or **`-AA`** control option to mention a control variable.)

Each control variable has a defined type: *integer*, *name*, or *name list*:

- An integer control variable has an integer value. For example, **`inllev`** can have the values *0*, *1*, *2*, *3*, *4*, or *5*.
- A name control variable has a name value or the value *null*. For example, **`inclpath`** can have the values *relative* or *absolute*.
- There are two name-list control variables, **`inline`** and **`volatile`**. These control variables can have several name values, one name value, or the value *null*.

Pragma directives are source-code statements that temporarily change the values of certain control variables. The individual explanations of paragraph 3.4 note which control variables are subject to pragma changes, as well as how long a pragma change lasts. For example, if a pragma directive changes the value of control variable **`rsave`**, the change lasts to the end of the file. If a pragma directive changes the value of control variable **`unroll`**, the change lasts to the end of the loop.

Paragraph 3.3 explains the syntax for pragma directives.

NOTE

Pragma directives *cannot* change control-variable values set via the **`-AA`** control option.

3.2 ASSIGNING CONTROL VARIABLE VALUES

To assign values to control variables, add individual assignments to **`-A`** or **`-AA`** control options of the compiler command line. You may use either option several times in the command line; each use may include one or more individual assignments.

Paragraphs 3.2.1 through 3.2.4 explain the basic syntax for assigning values to control variables. But the compiler accepts several alternative syntaxes; you may use any syntax that you find convenient, and you may mix syntaxes in your command line. Paragraph 3.5 explains the alternative syntaxes.

3.2.1 Assigning Integer Values

The general syntax of an individual assignment is the **name** of the control variable, an **equals sign**, and a **value**, without any spaces. For example,

```
-Agim=1
```

assigns the value 1 to control variable **`gim`**. This directs the compiler to do global instruction movement for basic blocks executed non-conditionally. Note that **`gim`** is usable in pragma directives; should a pragma directive give **`gim`** a different value, **`gim`** would regain the value 1 when the compiler begins compiling the next file.

The assignment

```
-AAgim=1
```

also assigns the value 1 to **`gim`**. But the **`-AA`** control option prevents pragma directives from changing the value.

To assign the second default value to a control variable, omit the equals sign and value from the individual assignment. For example, both

```
-Agim   and   -AAgim
```

give **gim** its second default value, 3.

NOTE

As paragraph 3.1 explained, not including a control-variable name in any **-A** or **-AA** option of the command line gives that control variable its first default value. For example, if you do not include **gim** in the command line, it receives the value 0.

3.2.2 Assigning Multiple Values

Usually, when you invoke the compiler, you assign specific values to several control variables, and mention (give second default values to) several other control variables. A convenient way to do this is to include several individual assignments in a **-A** or **-AA** control option, separating the assignments with commas.

For example, the control option below gives the respective values 1, 0, 0, and 2 to control variables **gim**, **inllev**, **retpts**, and **sched**. (Table 3-1 explains the significance of these values.)

```
-Agim=1,inllev=0,retpts=0,sched=2
```

You can assign second default values in the same way, as in this example:

```
-Aunroll=7,inllev,space,targ=505,gim,defvol=0,retpts
```

This **-A** control option assigns the respective values 7, 505, and 0 to control variables **unroll**, **targ**, and **defvol**. The same control option gives second default values (3, 2, 2, and 1, respectively) to control variables **inllev**, **space**, **gim**, and **retpts**.

You may put multiple assignments in **-AA** control options in the same way. The difference, of course, is that the **-AA** control option prevents pragma directives from changing the initial values of the control variables.

A single **-A** or **-AA** control option may become difficult to read if it includes all the value assignments for your command line. If so, you may divide the assignments among several **-A** or **-AA** control options in the command line, in any way that seems convenient.

NOTE

The compiler processes value assignments from left to right. Should a **-A** or **-AA** control option (or multiple options) include two or more value assignments for the same control variable, the compiler uses the right-most value assignment when it begins compilation.

3.2.3 Assigning Name Values

Control variables **c**, **char**, **comment**, **inclpath**, and **rsave** are the name control variables. Assign name values to these variables in the same way you assign values to integer variables. For example, the control option below gives respective values *mixed*, *unsigned*, and *multiple* to control variables **c**, **char**, and **rsave**:

```
-Ac=mixed,char=unsigned,rsave=multiple
```

Note that there are only two possible values for several of the name control variables, so that one is the first default value and the other is the second default value. This is true for **rsave**: its only possible values are *normal* (the first default value) and *multiple* (the second default value). This means that you can merely mention **rsave** to give it the value *multiple*, so that this control option makes the same value assignments as the previous one:

```
-Ac=mixed,char=unsigned,rsave
```

You may use the same **-A** or **-AA** control option to assign values of different types, as in this example:

```
-AAunroll=7,c=mixed,rsave,defvol=1,char=unsigned,retpts
```

This **-AA** control option assigns the respective values 7, *mixed*, 1, and *unsigned* to control variables **unroll**, **c**, **defvol**, and **char**. The same control option gives second default values (*multiple* and 1) to control variables **rsave** and **retpts**.

3.2.4 Assigning Name-List Values

Control variables **inline** and **volatile** are the name-list control variables. To assign multiple name values to either of these variables, separate the values with plus signs. For example, the control option below assigns values *press*, *temp*, *relhum*, and *dewpt* to control variable **volatile**:

```
-Avolatile=press+temp+relhum+dewpt
```

You may use the same **-A** or **-AA** control option to assign values to both **inline** and **volatile**. The same control option also may include value assignments for integer and name control variables.

The special name value **`%all`** denotes all the meaningful names for **`inline`** or **`volatile`**, so **`%all`** can shorten value assignments. For example, suppose that you want the compiler to consider all possible functions for inlining. Instead of scrutinizing all the possible functions of each file, you need only include this control option in the command line:

```
-Ainline=%all
```

For a subsequent compilation, suppose that you want to exclude the functions **`relhum`** and **`dewpt`** from consideration for inlining. Your command line should include this control option:

```
-Ainline=%all-relhum-dewpt
```

3.3 PRAGMA DIRECTIVE SYNTAX

As paragraph 3.1 explained, you can use pragma directives in source code to give temporary values to certain control variables. The duration of such a temporary value depends on the specific control variable.

Write pragma directives according to the syntax **pragma token**, **compiler token**, and **value assignment**:

```
#pragma moto assignment
```

where:

- Blanks must separate the three fields,
- **`#pragma`** (the pragma token) must start in column 1,
- **`#pragma`** is case sensitive,
- **`moto`** (the compiler token) is case sensitive,
- ***`assignment`*** represents any control-variable value assignment, identical to that of a command-line **`-A`** or **`-AA`** control option.
- The **`#pragma`** line *must not* include any other text (such as comments).

Assume, for example, that the compiler command line assigns the initial value 1 to control variable **`inllev`**, directing the compiler to inline small, simple functions. But you do not want inlining for a particular routine. Accordingly, you can write this pragma directive before the definition of a function:

```
#pragma moto inllev=0
```

This pragma directive gives the temporary value 0 to **`inllev`**, stopping inlining. The control variable regains the value 1 when compilation moves on to the next file.

If the same situation applied to control variable **sched**, you could use one pragma directive to temporarily stop both inlining and loop unrolling:

```
#pragma moto inllev=0,unroll=0
```

Follow the same pragma syntax for assigning name values, name-list values, and values of different types, just as you would in a **-A** control option. For example, this pragma directive assigns temporary values to control variables **inclpath** and **rsave**:

```
#pragma moto inclpath=relative,rsave=multiple
```

This pragma directive assigns temporary values to **inline**:

```
#pragma moto inline=min+max+avg+med+mean
```

Finally, note that the special name **%all**, for the name-list control variables **inline** and **volatile**, works the same way in pragma directives as it does in the command line.

3.4 CONTROL VARIABLE DEFINITIONS

Table 3-1 lists all control variables, explaining the meanings of their possible values. Note, in the Name column, that certain control variables are underlined. In order for such control variables to have an effect on compilation, the **-O** control option must have the value 3 or greater.

In the Values column, the footnote indicators **(1)** and **(2)** identify the first and second default values, respectively.

Paragraphs 3.4.1 through 3.4.26 explain each control variable individually.

Table 3-1. Control Variable Directory

Name	Values	Effect or Meaning
asm	0	Do *not* allow inline assembly pseudo-functions. (Default if C value is ANSI.)
	1(1)(2)	Process asm and other inline assembly pseudo-functions.
c	knr	Compile per K&R C definition.
	ansi(2)	Compile per standard ANSI C definition.
	relaxed(1)	Compile per least-strict ANSI C definition.
char	unsigned(1)	Make default char type unsigned.
	signed(2)	Make default char type signed.
comment	none(1)	Do *not* add scheduling comments to **.s** files.
	schedule(2)	Add basic block scheduling comments to **.s** files.
defvol	0(1)	Do not consider any variables volatile unless they are so declared.
	1	Treat all global variables as volatile, unless they are declared non-volatile.
	2(2)	Treat all off-stack variables as volatile, all pointers are type *pointer to volatile*.
diag	0	Output error and fatal-error diagnostic messages.
	1(1)	Output warning, error, and fatal-error diagnostic messages.
	2(2)	Output remark, warning, error and fatal-error diagnostic messages.
g	0(1)	Do *not* include debugging information in the assembly or object file.
	1(2)	Include debugging information in the assembly or object file.
	2	Include in the assembly or object file comments that reference lines of source code.
gim	0(1)	Do *no* global instruction movement.
	1	Do global instruction movement for basic blocks executed non-conditionally.
	2(2)	Do global instruction movement for all possible basic blocks.

Table 3-1. Control Variable Directory (cont.)

Name	Values	Effect or Meaning
inclpath	relative(1)	Search first for include files in the directory that contains the #include statement's file.
	absolute(2)	Search first for include files in the directory that contains the top-level source file.
inline	none(1)	Do not consider any functions for inlining.
	(user specified)	Consider listed functions for inlining.
	%all(2)	Consider all functions for inlining.
inllev	0(1)	Do *not* inline functions.
	1	Inline small, simple functions.
	2	Inline functions slightly larger or more complex.
	3(2)	Inline functions still larger or more complex.
	4	Inline functions larger or more complex yet.
	5	Inline the largest or most complex functions possible.
ipa	0(1)	Do not perform interprocedural analysis.
	1(2)	Perform interprocedural analysis.
memlimit	0(1)(2)	Do requested optimizations.
	i	Do the degree of requested optimizations that does not exceed the memory limit of i megabytes.
nofp	0	Allow floating-point loads and stores of non-floating-point data.
	1(1) (2)	Do not allow floating-point loads and stores of non-floating-point data.
pid	0(1)	Do not make data references position independent.
	1(2)	Make all data references position independent.
pic	0(1)	Do not make code position independent.
	1(2)	Make code position independent.
quit	0(1)	Quit for error or fatal-error deviations.
	1(2)	Quit for warning, error, or fatal-error deviations.
	2	Quit for remark, warning, error, or fatal-error deviations.

Table 3-1. Control Variable Directory (cont.)

Name	Values	Effect or Meaning
retpts	0	Allow multiple return points for compiled routines.
	1(1) (2)	Compile code so each routine has just one return point.
rsave	normal(1)	Use one instruction per general purpose register to save or restore registers.
	multiple(2)	Use load/store multiple word instructions to save or restore registers.
rosda_alloc	0	Do not allocate any globally visible external constant scalars in the **.sdata2** section.
	1—32768	Allocate to the **.sdata2** section constant scalars of this many or fewer bytes. (8 is the first and second default value.)
sched	0(1)	Do *no* instruction scheduling.
	1	Do instruction scheduling only during pass one.
	2(2)	Do instruction scheduling during pass one and pass two.
sda_alloc	0	Do not allocate any globally visible external scalars in the **.sdata** section.
	1—32768	Allocate to the **.sdata** section constant scalars of this many or fewer bytes. (8 is the first and second default value.)
space	0	Always favor faster code over smaller code.
	1(1)	Balance considerations of faster code versus smaller code.
	2(2)	Favor smaller code over faster code.
targ	505(1)(2)	Compile code to be run on a 505 MCU.
	601	Compile code to be run on a 601 MCU.
	603	Compile code to be run on a 603 MCU.
	604	Compile code to be run on a 604 MCU.
unroll	0(1)	Do *no* loop unrolling.
	1(2)	Do loop unrolling under automatic control.
	2 or more	Do loop unrolling this many times for every possible loop.
volatile	(user specified)	Consider listed variables volatile.

asm ASM Enable/Disable

3.4.1 ASM Enable/Disable (`asm`)

The **`asm`** control variable enables or disables processing of the inline assembly pseudo-functions: **asm**, **__asmul**, and **__asmd**. Paragraph 3.6 gives more information about these pseudo-functions.

Syntax: **`asm=0|1`**

where

0 Do *not* process inline assembly pseudo-functions. (Default if control variable **`c`** has the value *ansi*.)

1 Process inline assembly pseudo-functions. (**First and second default value**, if **`c`** does not have the value *ansi*.)

Usable in pragmas: no

Related control variables: **`c`**

Related control options: none

c C Dialect

3.4.2 C Dialect (c)

The **c** control variable sets the C-language dialect that the compiler is to compile. Motorola recommends the ANSI C standard, that is, the *ansi* value of *dialect*. Appendix B describes each dialect.

Syntax: **c=*dialect***

where ***dialect*** is

knr	Compile per the Kernighan and Ritchie definition of C.
ansi	Compile per the ANSI C standard; Motorola recommends this dialect value. (**Second default value**)
relaxed	Compile per the least-strict ANSI C definition, as explained above. (**First default value**)

Usable in pragmas: no

Related control variables: **asm**

Related control options: **-K**

char Character Type

3.4.3 Character Type (`char`)

The **`char`** control variable sets the default character type.

Standard ANSI C lets you declare variables of type *char* without specifying whether such variables are signed or unsigned. The value of **`char`** makes this specification for Motorola C.

Syntax: **`char=unsigned|signed`**

where

unsigned	Make the default character type unsigned. (**First default value**)
signed	Make the default character type signed. (**Second default value**)

Usable in pragmas: no

Related control variables: none

Related control options: none

comment Scheduling Comments

3.4.4 Scheduling Comments (`comment`)

The **`comment`** control variable enables or disables scheduling comments to the assembly-code (**`.s`**) file.

Syntax: **`comment=none|schedule`**

where

none	Do *not* add scheduling comments. (**First default value**)
schedule	Add basic block scheduling comments. (**Second default value**)

Usable in pragmas: no

Related control variables: **`sched`**

Related control options: none

defvol Default Volatile Variables

3.4.5 Default Volatile Variables (`defvol`)

The **`defvol`** control variable determines which variables the compiler considers volatile.

Syntax: **`defvol=`*`i`***

where ***`i`*** is

0 Do *not* consider any variables volatile, unless they are so declared (via the **`volatile`** control variable or the **`volatile`** C keyword). (**First default value**)

1 Treat all global and externally visible variables as volatile.

2 Treat all off-stack variables as volatile; treat all pointers as type pointer-to-volatile. (**Second default value**)

NOTES

For source code written in K&R C, the **`defvol`** control variable is one way to designate variables volatile. (Using the **`volatile`** control variable usually is a better way to designate variables volatile.)

If your source program includes volatile objects, it must declare them to be type **volatile**. If any pointers may reference volatile objects, the source program must declare the pointers to be type **pointer to volatile**.

If your source code includes volatile variables, your code must identify such variables to the compiler. You may use the control variable **`defvol`** to declare classes of variables volatile.

Assume that source code reads a variable, starts an asynchronous input operation on the variable, waits for the operation to end (by repeatedly checking a status-bit value), and then reads the variable again. Although the source code does not change the variable or the status bit, the asynchronous operation can change both. This means that the variable and the status bit are *volatile*.

To an optimizing compiler, the second read seems redundant: no code executing between the reads modifies the variable. Accordingly, the compiler removes the second read, using the value from the first read. This optimization would make the compiled program fail.

defvol Default Volatile Variables

This asynchronous I/O example shows how optimization can conflict with volatile objects. Other program features that can make memory objects volatile include multi-process shared memory, trap handlers, and special registers.

It is impractical to avoid source code that may contain volatile objects. But there are some practical ways to prevent the optimization-and-volatile-object conflict from causing program failure:

1. Explicitly declaring all volatile objects to be of type **volatile**, and declaring all pointers to volatile objects to be of type **pointer to volatile**. For instructions on making these declarations, consult your C-language documentation.
2. Using the control variable **`defvol`** to designate classes of variables volatile and classes of pointers as pointers to volatile objects.
3. Using the control variable **`volatile`** to designate specific variables volatile and specific pointers as pointers to volatile objects.

If you designate variables volatile or pointers as pointing to volatile objects, per options 2, 3, or 4 above, the Motorola compiler avoids the optimization-and-volatile-object conflict by preserving the sequence of references to volatile objects.

Usable in pragmas: yes

NOTE

If your source file includes any **`#pragma moto defvol`** statement, the statement must not be within the body of any function. The pragma's effect lasts until the compiler encounters another **`#pragma moto defvol`** statement, or until the compiler arrives at the end of the file.

Example: Assume that you will enable optimizations for compiling this example C source code. (That is, assume that the **`mecc`** command line will include the option **`-O3`** or **`-O4`**.)

```
int counter;              /* Globally visible variable. */
/* Treat all global and externally visible variables as volatile: */
#pragma moto defvol=1
```

defvol Default Volatile Variables

```
void function1 (void)
{
  counter = 10;            /* This assignment won't be optimized out. */
  counter = 20;
}
/* Do not treat global and externs as volatile: */
#pragma moto defvol=0

void function2 (void)
{
  counter = 30;            /* This assignment will be optimized out. */
  counter = 40;
}

void function3 (void)
{
#pragma moto defvol=1
  counter = 50;            /* This assignment won't be optimized out. */
  counter = 60;
}

void function4 (void)
{
  counter = 70;            /* This assignment won't be optimized out. */
  counter = 80;
}
```

Related control variables: `volatile`

Related control options: none

diag Diagnostic Messages

3.4.6 Diagnostic Messages (`diag`)

The **`diag`** control variable works with control variable **`quit`** to determine how the compiler responds to deviations from the C-language standard. In particular, **`diag`** determines which categories of diagnostic messages the compiler outputs. In command line mode, when the compiler does output such messages, it outputs them to the file **`stdout`**.

Syntax: **`diag=`*`i`***

where ***`i`*** is

0 Output error and fatal-error diagnostic messages.

1 Output warning, error, and fatal-error diagnostic messages. (**First default value**)

2 Output remark, warning, error, and fatal-error diagnostic messages. (**Second default value**)

The compiler evaluates each deviation from standard C for acceptability, issuing a corresponding internal diagnostic message. The five categories of these messages are:

- *Standard* — for acceptable language extensions.
- *Remark* — for unconventional, but acceptable usage.
- *Warning* — for questionable usage that the compiler nevertheless accepts.
- *Error* — for syntax or semantic violations. (The compiler may continue compilation, in order to identify other problems, but does not produce object code.)
- *Fatal error* — for violations so severe that compilation cannot continue.

Usable in pragmas: no

Related control variables: **`quit`**

Related control options: **`-w`**

g Debugging Information

3.4.7 Debugging Information (g)

The **g** control variable determines whether the compiler includes symbolic debugging information (source annotation, symbols, and line numbers) in the assembly file that the compiler generates.

Syntax: **g=*i***

where ***i*** is

0 Do *not* include debugging information in the assembly file. (**First default value**)

1 Include debugging information in the assembly file. (**Second default value**)

2 Include in the assembly file comments that reference lines of the source code. (Do not include debugging information.)

Usable in pragmas: no

Related control variables: none

Related control options: **-g**

gim Global Instruction Movement

3.4.8 Global Instruction Movement (`gim`)

The **`gim`** control variable determines the level of global instruction movement the compiler is to perform.

Global instruction movement is moving instructions across basic blocks, to improve instruction scheduling.

Syntax: **`gim=`*`i`***

where ***`i`*** is:

0 Do *not* perform global instruction movement. (**First default value**)

1 Perform global instruction movement for basic blocks executed non-conditionally.

2 Perform global instruction movement for all possible blocks. (**Second default value**)

NOTE

The compiler implements global instruction movement only for optimization levels 3 or 4 (**`-O3`** or **`-O4`**). If you specify a lower optimization level and your command line includes **`-Agim=1`** or **`-Agim=2`**, the compiler ignores your request for global instruction movement.

Usable in pragmas: no

Related control variables: none

Related control options: **`-O`**

inclpath Include Path

3.4.9 Include Path (`inclpath`)

The **`inclpath`** control variable determines where the compiler first looks for a file that an **`#include`** statement specifies (via a relative filename, in quotes).

All C implementations search for files named in **`#include`** statements, but they may not start such searches in the same directory. (Standard ANSI C does not specify the starting directory.) The **`inclpath`** value resolves this inconsistency.

Syntax: **`inclpath=relative|absolute`**

where

relative	Begin the search in the directory that contains the **`#include`** statement's file. (**First default value**)
absolute	Begin the search in the directory that contains the top-level source file. (**Second default value**)

Usable in pragmas: no

Related control variables: **`c`**

Related control options: **`-I`**

inline Inline Functions

3.4.10 Inline Functions (`inline`)

The **`inline`** control variable contains the list of functions which the compiler is to consider for inlining.

Inlining is moving the instructions of a subroutine to the calling routine, in order to eliminate the overhead of calling the subroutine and returning from the subroutine.

This control variable works in conjunction with **`inllev`**. The first default value is *none*: do not consider any functions for inlining. The second default value is **%all**: consider all functions for inlining.

Syntax: **`inline=name1+name2+...`**

where **`name1+name2+...`** is the list of function names. **`%inline`** means the current list.

Usable in pragmas: yes

NOTE

If your source file includes any **`#pragma moto inline`** statement, the statement must not be within the body of any function. The pragma's effect lasts until the compiler encounters another **`#pragma moto inline`** statement, or until the compiler arrives at the end of the file.

Example: Assume that the **`mecc`** command line will include options **`-S -O -Ainllev=1`**.

```
/* This #pragma tells the compiler to consider all functions as
 * candidates for inlining: */
#pragma moto inline

long aplusb(long a, long b);

int main()
{
  long a, b, c;

  a = 100;
  b = 50;

  c = aplusb(a,b);      /* This will be inline code. */
  return (c);
}
```

inline Inline Functions

```
/* Remove aplusb() from consideration for inlining. */
#pragma moto inline=%inline-aplusb
int function1()
{
  long a, b, c;

  a = 1000;
  b = 500;

  c = aplusb(a,b);      /* This will be a procedure call. */
  return (c);
}
/* Put aplusb() back on the list of functions to be inlined. */
#pragma moto inline=%inline+aplusb
int function2()
{
  long a, b, c;

  a = 10;
  b = 5;

  c = aplusb(a,b);      /* This will be inline code. */
  return (c);
}

long aplusb(long a, long b)
{
  return(a+b);
}
```

Related control variables: **`inllev`**

Related control options: none

inllev Enable/Disable Inlining

3.4.11 Enable/Disable Inlining (inllev)

The **`inllev`** control variable sets the degree of inlining the compiler is to perform on the functions that control variable **`inline`** contains.

Two factors determine whether the compiler actually inlines a function: the complexity of the function and the context into which the function may be inlined.

Syntax: **`inllev=`*`i`***

where ***`i`*** is

- *0* Do *not* inline functions. (**First default value**)
- *1* Inline small, simple functions.
- *2* Inline functions slightly larger or more complex.
- *3* Inline functions still larger or more complex. (**Second default value**)
- *4* Inline functions larger or more complex yet.
- *5* Inline the largest or most complex functions possible.

Usable in pragmas: yes

NOTES

If your source file includes any **`#pragma moto inllev`** statement, the statement must not be within the body of any function. The pragma's effect lasts until the compiler encounters another **`#pragma moto inllev`** statement, or until the compiler arrives at the end of the file.

The compiler does not let a pragma specify an **`inllev`** value if you invoke the compiler with an optimization level less than 2 (**`-O0`** or **`-O1`**). Similarly, the compiler does not let a pragma specify an **`inllev`** value if the **`mecc`** command line does not specify an **`inllev`** value greater than 0 (for example, **`-Ainllev=1`**).

inllev Enable/Disable Inlining

Example: Assume that the **mecc** command line will include options **-O -Ainllev=1**.

```
/* Inline all functions possible. */
#pragma moto inline

long aplusb(long a, long b);
long aminusb(long a, long b);

/* Turn off inlining from here ... */
#pragma moto inllev=0
int main()
{
  long c;
  c = aminusb(100,50);  /* Will not be inlined. */
  return (c);
}
int function1()
{
  long c;
  c = aplusb(1000,500); /* Will not be inlined. */
  return (c);
}
/* Turn level 1 inlining back on from here. */
#pragma moto inllev=1
int function2()
{
  long c;
  c = aplusb(10,5);     /*Will be inlined. */
  return (c);
}

long aplusb(long a, long b)
{
  return(a+b);
}
long aminusb(long a, long b)
{
  return(a-b);
}
```

Related control variables: inline

Related control options: -O

ipa Interprocedural Analysis

3.4.12 Interprocedural Analysis (`ipa`)

The **`ipa`** control variable tells the compiler whether to perform interprocedural analysis, and whether to apply optimizations across multiple procedures.

Syntax: **`ipa=0|1`**

where

0 Do not perform interprocedural analysis: optimize each procedure individually. (**First default value**)

1 Perform interprocedural analysis: take multiple procedures into account as part of optimization. (**Second default value**)

Usable in pragmas: no

Related control variables: none

Related control options: none

memlimit Memory Limit

3.4.13 Memory Limit (`memlimit`)

The **`memlimit`** control variable lets you increase the memory allocation for compiler optimizations. If the compiler needs more memory than the 32 megabyte default to carry out all the optimizations you request, the compiler performs the optimizations that fit within the limit, then issues a message estimating the memory necessary for all the requested optimizations.

To increase the amount of memory the compiler allocates, enter this control variable with an integer value: the compiler then can allocate that many megabytes of memory. In all situations, the compiler performs the greatest degree of optimization possible within the memory limit.

Syntax: **`memlimit=0|`*`i`***

where

- 0 Perform all requested optimizations, using the default 32 megabytes of memory. If that is not possible, perform as many optimizations as possible, then issue a message estimate of memory required for all the requested optimizations. (**First and second default values**)
- *i* An integer greater than 32: allocate this many megabytes; perform all requested optimizations possible in this space.

Usable in pragmas: no

Related control variables: none

Related control options: none

nofp No FP Moves

3.4.14 No FP Moves (`nofp`)

The **`nofp`** control variable determines whether the compiler can use floating-point loads and stores to move non-floating-point data. If the compiler can do so, fewer instructions are necessary to move a given amount of data.

Syntax: **`nofp=0|1`**

where

0 Allow the use of floating-point loads and stores to move non-floating-point data.

1 Do not allow the use of floating-point loads and stores to move non-floating-point data. (**First and second default values**)

Usable in pragmas: no

Related control variables: none

Related control options: none

pic Position Independent Code

3.4.15 Position Independent Code (pic)

The **pic** control variable determines whether the compiler generates position independent (relocatable) code. If the compiler does, at run time you can move the **.text** section to an address different from the address it received during linking. Instructions may reference symbols in the **.text** section, but this does not require the symbols to be at particular addresses.

Syntax: **pic=0|1**

where

0 Do not make code position independent. (**First default value**)

1 Make code position independent. (**Second default value**)

NOTE

The extended conformance level of the PowerPC Embedded Application Binary Interface (EABI) standard includes a definition of position independent code. If you use the **pic** control variable with the value 1, the compiler generates position independent code that does *not* comply with the EABI definition.

Specifying code to be position independent, imposes an initialization restriction regarding any global or static variable that is a pointer to a function. Your declarations cannot give such a variable an initial value that specifies a function address.

If an executable consists of multiple object files or library files, you must compile each object or library file with the **pic** value 1, in order for the executable to be position-independent

Appendix C explains library routines, some of which support the position-independent code that you produce via the **pic** control variable.

Usable in pragmas: no

Related control variables: **pid**

Related control options: **none**

pid Position Independent Data

3.4.16 Position Independent Data (pid)

The **pid** control variable determines whether the compiler generates position independent (relocatable) data. If the compiler does, at run time you can move program sections containing data to addresses different from the addresses they received during linking. Instructions may reference data in such sections, but this does not require the data to be at particular addresses.

Syntax: **pid=0|1**

where

0 Do not make data position independent. (**First default value**)

1 Make data position independent. (**Second default value**)

Specifying data to be position independent only affects global and static variables. The object code creates variables with local extent (automatic variables) upon entry to a function or block; upon exiting the function or block, the object code destroys automatic variables.

Specifying data to be position independent imposes an initialization restriction regarding any global or static pointer variable. Your declarations cannot give such a position independent pointer variable an initial value that specifies the address of another position independent variable (including string constants).

NOTES

The extended conformance level of the PowerPC Embedded Application Binary Interface (EABI) standard includes a definition of position independent code. If you use the **pid** control variable with the value 1, the compiler generates position independent code that does *not* comply with the EABI definition.

With regard to position independence, the compiler considers the sections **.data**, **.bss**, and **.sbss** to be a single segment of read/write data. Any run-time relocation of read/write data must move all read/write sections as a unit, by the same amount. Similarly, the compiler considers the sections **.rodata** and **.sdata2** to be a single segment of read-only data; a run-time relocation of read-only data must move all the read-only sections as a unit.

pid Position Independent Data

NOTE

The computer addresses the read/write data relative to the SDA register and the read-only data relative to the SDA2 register. This enables the object program to update these registers if there is a run-time relocation of data. Chapter 7 of the MEAS assembler user's manual gives more information about the SDA and SDA2 registers.

If an executable consists of multiple object files or library files, you must compile each object or library file with the **`pid`** value 1, in order for the executable's data to be position independent.

Appendix C explains library routines, some of which support the position-independent code that you produce via the **`pid`** control variable.

Usable in pragmas: no

Related control variables: **`pic`**

Related control options: none

quit Quit for Diagnostics

3.4.17 Quit for Diagnostics (`quit`)

The **`quit`** control variable works with control variable **`diag`** to determine how the compiler responds to deviations from the C-language standard. In particular, **`quit`** determines the deviation severity level at which the compiler quits (that is, does not try to produce a **`.s`** file).

If the compiler quits based on a remark, warning, or error, it continues compilation activity, trying to find any additional deviations.

Note that the basis for quitting is the deviation, not the corresponding diagnostic message. For example, if the **`quit`** value is 2 and the **`diag`** value is 0 when the compiler finds a remark deviation, compilation stops but the compiler does not output a remark message.

(Paragraph 3.4.6 explains the categories of MECC diagnostic messages.)

Syntax: **`quit=`*`i`***

where ***`i`*** is

- 0 Quit based on error or fatal-error deviations; output corresponding diagnostic messages. (**First default value**)
- 1 Quit based on warning, error, or fatal-error deviations; output corresponding diagnostic messages per the **`diag`** value. (**Second default value**)
- 2 Quit based on remark, warning, error, or fatal-error deviations; output corresponding diagnostic messages per the **`diag`** value.

Usable in pragmas: no

Related control variables: **`diag`**

Related control options: **`-w`**

retpts Return Points

3.4.18 Return Points (`retpts`)

The **`retpts`** control variable determines whether routines of the compiled code may have multiple return points. (A return point is a sequence of instructions that return program flow to the calling routine.) Code that includes multiple return points usually executes faster than equivalent code that has only one return point, even though the multiple-return-point code is larger. Accordingly, the compiler can add return points to improve performance. Additionally, certain other optimizations may create multiple return points from a single source-code return statement, further enhancing execution speed.

Syntax: **`retpts=0|1`**

where

0 Allow multiple return points for compiled routines.

1 Compile code so that each routine has just one return point. (**First and second default value**)

Usable in pragmas: yes

NOTE

If your source file includes any **`#pragma moto retpts`** statement, the statement must not be within the body of any function. The pragma's effect lasts until the compiler encounters another **`#pragma moto retpts`** statement, or until the compiler arrives at the end of the routine.

Example: This example is compiled with the **`-O`** option.

```
int a, b, c;
/* Suppress collapsing of multiple return points into a common */
/* return point. */
#pragma moto retpts=0
void main (int arg1, int arg2)
{
  if (arg1 > 0) {
    a = 12;
    b = 0;
    c = 10;
    return;
  }
  if (arg1 < 0) {
    a = b = c = 99;
    return;
  }
}
/* Collapsing multiple return points into a common return point. */
#pragma moto retpts=1
void other (int arg1, int arg2)
```

retpts Return Points

```
{
  if (arg1 > 0) {
    a = 12;
    b = 0;
    c = 10;
    return;
  }
  if (arg1 < 0) {
    a = b = c = 99;
    return;
  }
}
```

Related control variables: `rsave`

Related control options: none

rosda_alloc Read-Only SDA Allocation

3.4.19 Read-Only Small Data Area Allocation (`rosda_alloc`)

The **`rosda_alloc`** control variable directs the compiler to allocate constant global variables in the **`.sdata2`** area. The **`rosda_alloc`** value is the maximum number of bytes such a variable may occupy in that area. If the size of the variable exceeds the **`rosda_alloc`** value, the compiler instead allocates the variable in the **`.rodata`** area.

The **`rosda_alloc`** value also affects how the compiler addresses *external* constant global variables. If the **`rosda_alloc`** value is 8, the compiler generates code to access all constant externals of 8 or fewer bytes as if they were allocated in the **`.sdata2`** area. The compiler assumes that any external constants larger than 8 bytes must be in the **`.rodata`** area.

Allocating constant global variables in the **`.sdata2`** area makes it possible to address them using a 16-bit offset and the SDA2 register. (Chapter 7 of the MEAS assembler user's manual gives more information about the SDA2 register.) This means that in most cases loading a value requires only one instruction, leading to smaller, faster compiled code.

NOTE

If an executable consists of multiple object files or library files, you should compile each object or library file with the same **`rosda_alloc`** value. Doing otherwise could prevent linking or cause incorrect program execution.

Syntax: **`rosda_alloc=`*`i`***

where ***`i`*** is

0	Do *not* allocate constant global variables to the **`.sdata2`** area; allocate all constant global variables to the **`.rodata`** area.
1— 32768	Allocate to the **`.sdata2`** area constant global variables whose size is less than or equal to this value; allocate larger constant global variables to the **`.rodata`** area. (**The first and second default value is 8.**)

Usable in pragmas: no

Related control variables: **`sda_alloc`**

Related control options: none

rsave Register Save

3.4.20 Register Save (`rsave`)

The **`rsave`** control variable determines what instructions the compiler uses to save or restore registers in function prologs and epilogs.

Syntax: **`rsave=normal|multiple`**

where

normal — Use one instruction per general purpose register to save or restore registers. (**First default value**)

multiple — Use a load/store multiple word instruction to save or restore general purpose registers. (**Second default value**)

Usable in pragmas: yes

NOTE

If your source file includes any **`#pragma moto rsave`** statement, the statement must not be within the body of any function. The pragma's effect lasts until the compiler encounters another **`#pragma moto rsave`** statement, or until the compiler arrives at the end of the file.

Example: This example has no visible results unless the **`-O`** option value is 3 or greater.

```
void external_func (int *, int *)
/* Use a single stmw (store multiple word) and single lmw (load
 * multiple word) to save/restore general purpose registers in
 * function prolog/epilong sequences.
 */
#pragma moto rsave=multiple
void main(void)
{
  int a, b, c, d;

  a = 22;
  b = 44;
  for (c = 10; c < 100; c++)
    for (d = 0; d < 10; d++)
      printf ("alpha", a, "beta", b, "gamma", "delta", c, d);
  external_func(&a, &b);
}
/* Use multiple stw (store word) and lwz (load word) instructions to
 * save/restore general purpose registers in function prolog/epilog
 * sequences.
 */
#pragma moto rsave=normal
void other(void)
```

rsave Register Save

```
{
  int a, b, c, d;

  a = 22;
  b = 44;
for (c = 10; c < 100; c++)
  for (d = 0; d < 10; d++)
     printf (“alpha”, a, “beta”, b, “gamma”, “delta”, c, d);
  external_func(&a, &b);
}
```

Related control variables: `retpts`

Related control options: none

sched Instruction Scheduling

3.4.21 Instruction Scheduling (`sched`)

The **`sched`** control variable determines whether and when the compiler does instruction scheduling.

Instruction scheduling is reordering instructions to take advantage of the processor's capabilities. In particular, if the processor can execute certain instructions in parallel, instruction scheduling can reduce program execution time significantly.

The **`-O`** control option assigns a value to **`sched`** automatically: **`-O0`** or **`-O1`** gives **`sched`** the value 0; **`-O2`**, **`-O3`**, or **`-O4`** gives **`sched`** the value 1.

Syntax: **`sched=`*`i`***

where ***`i`*** is

0 Do *no* instruction scheduling. (**First default value**)

1 Do instruction scheduling only during pass one of register allocation.

2 Do instruction scheduling during pass one *and* during pass two of register allocation. (**Second default value**)

Usable in pragmas: no

Related control variables: **`comment`**

Related control options: **`-O`**

sda_alloc SDA Allocation

3.4.22 Small Data Area Allocation (sda_alloc)

The **sda_alloc** control variable directs the compiler to allocate global variables in the **.sdata** and .**sbss** areas. The **sda_alloc** value is the maximum number of bytes such a variable may occupy in that area. If the size of the variable exceeds the **sda_alloc** value, the compiler instead allocates the variable in the **.data** or **.bss** area.

The **sda_alloc** value also affects how the compiler addresses *external* non-constant global variables. If the **sda_alloc** value is 8, the compiler generates code to access all externals of 8 or fewer bytes as if they were allocated in the **.sdata** or **.sbss** area. The compiler assumes that any external variables larger than 8 bytes must be in the **.data** or **.bss** area.

Allocating global variables in the **.sdata** or **.sbss** area makes it possible to address them using a 16-bit offset and the SDA register. (Chapter 7 of the MEAS assembler user's manual gives more information about the SDA register.) This means that in most cases loading a value requires only one instruction, leading to smaller, faster compiled code.

NOTE

If an executable consists of multiple object files or library files, you should compile each object or library file with the same **sda_alloc** value. Doing otherwise could prevent linking or cause incorrect program execution.

Syntax: **sda_alloc=*i***

where ***i*** is

0	Do *not* allocate globals to the **.sdata** or .**sbss** area; allocate all globals to the **.data** or **.bss** area.
1— 32768	Allocate to the **.sdata** area global variables whose size is less than or equal this value; allocate larger global variables to the **.data** area. (**The first and second default value is 8.**)

Usable in pragmas: no

Related control variables: **rosda_alloc**

Related control options: none

space Limit Code Space

3.4.23 Limit Code Space (`space`)

The **`space`** control variable directs the compiler to favor smaller executable code instead of faster executable code. (Certain code sequences may execute faster in a PowerPC processor, even though they take up more space than comparable sequences.)

Syntax: **`space=`*`i`***

where ***`i`*** is

0 Favor faster-executing code over smaller code.

1 Balance considerations of code execution speed versus code size. (**First default value)**

2 Favor small code over faster-executing code. (**Second default value**)

Usable in pragmas: no

Related control variables: none

Related control options: **`-O`**

targ Target Processor

3.4.24 Target Processor (`targ`)

The **`targ`** control variable tells the compiler the processor on which the program is to be run. This knowledge enables the compiler to apply certain processor-specific enhancements to its optimizations.

Syntax: **`targ=`*`i`***

where ***`i`*** is

505 Apply optimization enhancements specific for a 505 target processor. (**First and second default value**)

601 Apply optimization enhancements specific for a 601 target processor.

603 Apply optimization enhancements specific for a 603 target processor.

604 Apply optimization enhancements specific for a 604 target processor.

Usable in pragmas: no

Related control variables: none

Related control options: none

unroll Loop Unrolling

3.4.25 Loop Unrolling (`unroll`)

The **`unroll`** control variable determines the level of loop unrolling the compiler is to perform.

Loop unrolling is replicating several times the code of innermost loops. This means that the system tests fewer conditions, reducing program execution time.

The **`-O`** control option assigns a value to **`unroll`** automatically: **`-O0`** or **`-O1`** gives **`unroll`** the value *0*; **`-O2`**, **`-O3`**, or **`-O4`** gives **`unroll`** the value *1*.

Syntax: **`unroll=`*`i`***

where ***`i`*** is

0	Do *no* loop unrolling. (**First default value**)
1	Do loop unrolling under automatic control. (**Second default value**)
2 or more	Do loop unrolling for every possible loop this many times.

Usable in pragmas: yes

NOTE

You may place a **`#pragma moto unroll`** statement anywhere in a C source file. The pragma's effect lasts until the compiler encounters another **`#pragma moto unroll`** statement, or until the compiler arrives at the end of the file.

Example: For this example, assume that the **`-O`** option value is 2 or greater.

```
#define AS 20
main()
{
  int 1;
  int iA[AS];
  float aA[AS];
  for (i = 0; i < AS; i++) { /* This loop gets unrolled. */
    iA[i] = 0;
  }
#pragma moto unroll=0
  for (i = 0; i < AS; i++) { /* This loop will not be unrolled. */
    aA[i] = 1.0;
  }
}
same_func()
```

unroll Loop Unrolling

```
{
  int i;
  int iA[AS];
  float aA[AS];
  for (i = 0; i < AS; i++) { /* This loop will not be unrolled. */
    iA[i] = 0;
  }
#pragma moto unroll=1
  for (i = 0; i < AS; i++) { /* This loop will be unrolled. */
    aA[i] = 1.0;
  }
}
```

Related control variables: none

Related control options: **`-O`**

volatile Volatile Variables

3.4.26 Volatile Variables (`volatile`)

The **`volatile`** control variable contains the list of variables which the compiler is to consider volatile. There is no default value for **`volatile`**.

Syntax: **`volatile=name1+name2+...`**

where **`name1+name2+...`** is the list of variables

The special name **`%volatile`** lets you redefine the list of volatile variables in terms of the most recent definition, plus or minus other variable names. For example, this command-line argument:

```
-Avolatile=xxx+yyy
```

declares variables **`xxx`** and **`yyy`** volatile. Later, in the same command line, you could declare **`zzz`** volatile, as well, by including this additional argument:

```
-Avolatile=%volatile+zzz
```

Use the minus sign to remove a variable from the list of volatile variables. For example, suppose that you included this third argument in the same command line:

```
-Avolatile=%volatile-yyy
```

When the compiler processes this argument, it would remove **`yyy`** from the list of volatile variables, leaving **`xxx`** and **`zzz`** still volatile.

The special name **`%all`** tells the compiler to consider *all* the variables volatile, including global variables, local static variables, and automatic stack variables. For example, this command-line argument:

```
-Avolatile=%all-www
```

tells the compiler to consider all the variables but **`www`** volatile. Use **`%all`** carefully, however, as identifying most variables volatile severely limits the compiler's ability to optimize your code.

Usable in pragmas: yes

NOTE

You must place a **`#pragma moto volatile`** statement *before* the declarations of variables to be declared volatile. Once the compiler encounters the definition of a variable, **`#pragma moto volatile`** statements cannot make the variable volatile.

volatile Volatile Variables

Example: For this example, assume that the command line includes the arguments **`-O -Avolatile=counter2`**.

```
/* Add counter to the list of volatile variables */
#pragma moto volatile=%volatile+counter
int counter;

/* Removed counter2 from the list or volatile variables */
#pragma moto volatile=%volatile-counter2
int counter2;
void function1 (void)
{
  counter = 10;
  counter = 20;            /* This store will not be optimized out. */
}
void function2 (void)
{
  counter = 30;
  counter = 40;            /* This store will not be optimized out. */
}
void function3 (void)
{
  counter = 50;
  counter = 60;            /* This store will not be optimized out. */
}
/* Try to make counter2 volatile again, This will not work. */
#pragma moto volatile=%volatile
void function4 (void)
{
  counter = 30;
  counter = 40;            /* This store will not be optimized out. */
  counter2 = 70;
  counter2 = 80;        /* This store will be optimized out. */
}
void function5 (void)
{
  counter = 70;
  counter = 80;            /* This store will be optimized out. */
  counter2 = 90;
  counter2 = 100;       /* This store will be optimized out. */
}
```

Related control variables: `defvol`

Related control options: none

3.5 ALTERNATE ASSIGNMENT SYNTAX

Paragraph 3.2 explained the basic syntax for assigning a value to a control variable: the **name**, an **equals sign**, and the **value**, as part of a **-A** or **-AA** control option. Provided there is no ambiguity, you can use any of these alternate patterns:

- the **name**, an **equals sign**, and the **value in parentheses**
- the **name** and the **value in parentheses** (without an equals sign)
- the **name** and the **value** (without an equals sign)

This means that any of these control options assigns the value *4* to control variable **inllev**:

```
-Ainllev=4    -Ainllev=(4)  -Ainllev(4)   -Ainllev4
```

You may even mix syntaxes for a control option that assigns several values:

```
-Aagim=(1), inllev(0), retpts=(0), unroll7, defvol=1, space1
```

But to prevent ambiguity, the syntax for a name control variable *must* have an equals sign or parentheses. Any of these control options assigns the value *ansi* to control variable **c**:

```
-Ac=ansi      -Ac=(ansi)    -Ac(ansi)
```

Value assignments for name-list control variables *must* have addition signs, but may use parentheses instead of the equals sign. Either of these examples assigns the same names to **volatile**:

```
-Avolatile=press+temp+relhum+dewpt
-Avolatile(press+temp+relhum+dewpt)
```

The compiler lets you assign multiple values, in any order, in mixed syntax, in the same **-A** or **-AA** control option. Accordingly, many possible control options assign exactly the same values, even though they differ in syntaxes and order.

3.6 INLINE ASSEMBLY PSEUDO-FUNCTIONS

The special pseudo-functions **asm( )**, **__asmul( )**, and **__asmd( )** let you embed PowerPC assembly language in your C source code anywhere that C allows a function call. This lets you access supervisor-level instructions without writing assembly-language programs.

You can use these pseudo-functions anywhere C allows a normal function call. (You may think of such a pseudo-function as another executable statement of your C program.)

NOTE

Use the inline assembly pseudo-functions carefully. Each entails significant potential for embedding assembly-language statements that could cause incorrect program execution. For example, you could easily overlook an interaction between the assembly instruction you embed into the code and the code the compiler generates.

Although these pseudo-functions are not part of standard ANSI C, there are two ways to make their use possible:

- Use the K&R or relaxed C dialect instead of ANSI C. The **c** control variable lets you specify either dialect; the -K option specifies the K&R dialect.
- Enable the pseudo-functions by specifying the value 1 for the **asm** control variable.

(If you use any of these pseudo-functions with standard ANSI C, the compiler will regard it as a user-supplied function, generating a call to a function named **asm( )**, **__asmul( )**, or **__asmd( )**.

3.6.1 asm( )

The **asm()** pseudo-function embeds the specified assembly-language instructions in your C source code. This pseudo-function does not have a return value.

Syntax: **asm("*instruction*")**

where *instruction* is one or more valid PowerPC assembly-language instructions, to be inserted verbatim into the **.s** file the compiler generates.

NOTES

The compiler cannot verify that the *instruction* value consists of valid PowerPC assembly-language instructions. Using the **asm( )** pseudo-function with any other instruction value can cause incorrect code.

The compiler does not know which assembly-language instructions you use in the **asm( )** pseudo-function, so the compiler cannot protect against any error that these instructions may cause by modifying registers.

Example: This program inserts the **eieio** instruction into the **.s** file:

```
extern void some_other_routine(int arg1, int arg2);
void main(void)
{
   int a, b;
    a = 12;

   asm("\teieio\t\t\t #inserted by asm() pseudo-function");
   b = 20;
   some_other_routine(a,b);
}
```

Note that the function argument contains tab formatting directives (**\t**) as well as the **eieio** instruction. Placing a leading tab and a trailing comment in this way makes it easy to find the inserted instruction in the **.s** file.

The corresponding part of the generated **.s** file would be code of this form:

```
.main:
#  start of prologue, stack size = 64
   mfspr             r0,1r
   stw               r0,0x8(sp)
   stwu              sp,-64(sp)
#  end of prologue
   addi              r3,r0,0xc
   stw               r3,0x38(sp)
   eieio             #inserted by asm() pseudo-function
   addi              r4,r0,0x14
```

```
stw             r4,0x3c(sp)
bl              .some_other_routine
nop
.
.
```

The **eieio** instruction appears between the assignments of value 12 to variable **a** and value 20 to variable **b**.

3.6.2 __asmul( )

The **__asmul( )** pseudo-function function embeds the specified assembly-language instructions in your C source code, and lets your inline assembly read and write C variables. This pseudo-function has a return value: the contents of **r3** (a value of type unsigned long).

Syntax: **unsigned long __asmul ("*instruction*", [*values*,...])**

where

unsigned long — indicates the return-value type. The return value is the value in register **r3**.

instruction — is one or more valid PowerPC assembly-language instructions, to be inserted verbatim into the **.s** file the compiler generates.

values — are as many as 8 non-floating-point argument values, and as many as 8 floating-point argument values. The compiler assigns the non-floating-point values to registers **r3** through **r10**; it assigns the floating-point values to registers **f1** through **f8**. Once in registers, these values are available to the instructions that make up the *instruction* value.

The compiler loads the *values* values into registers before it embeds the *instruction* instructions. This lets your inline assembly code operate on those values.

The **__asmul( )** pseudo-function is much like a function call to the compiler: the compiler saves values of volatile registers before executing the pseudo-function, and restores those register values after executing the pseudo-function. This means that the embedded instructions can use volatile registers without interfering with other code the compiler generates. Paragraph 5.2 gives additional information about register usage.

NOTES

The compiler cannot verify that the *instruction* value consists of valid PowerPC assembly-language instructions. Using the **__asmul( )** pseudo-function with any other instruction value can cause incorrect code.

The compiler loads *values* values into registers according to the parameter-passing conventions paragraph 5.4 explains.

The compiler does not know which assembly-language instructions you use in the **__asmul( )** pseudo-function, so the compiler cannot protect against any error that these instructions may cause by modifying registers.

Example:

```
{
   unsigned long a = 2;
   unsigned long b = 3;
   unsigned long c;

   c = __asmul("\
      andi    r3,r3,0xff  \n\
      andi    r4,r4,0xff  \n\
      add     r3,r3,r4    \n\", a, b);
}
```

In this example, the compiler passes arguments (local variables **a** and **b**) to **__asmul( )**. The pseudo-function loads them into **r3** and **r4**, per the parameter passing conventions. The assembly instructions operate on these values, leaving the result in **r3**. The pseudo-function returns the value in **r3**, assigning it to variable **c**.

3.6.3 __asmd( )

The **__asmd( )** pseudo-function function embeds the specified assembly-language instructions in your C source code, and lets your inline assembly read and write C variables. This pseudo-function has a return value: the contents of **f1** (a value of type double).

Syntax: `double __asmd("instruction", [values,...])`

where

double — indicates the return-value type. The return value is the value in register **f1**.

instruction — is one or more valid PowerPC assembly-language instructions, to be inserted verbatim into the **.s** file the compiler generates.

values — are as many as 8 non-floating-point argument values, and as many as 8 floating-point argument values. The compiler assigns the non-floating-point values to registers **r3** through **r10**; it assigns the floating-point values to registers **f1** through **f8**. Once in registers, these values are available to the instructions that make up the *instruction* value.

The compiler loads the *values* values into registers before it embeds the *instruction* instructions. This lets your inline assembly code operate on those values.

The **__asmd( )** pseudo-function is much like a function call to the compiler: the compiler saves values of volatile registers before executing the pseudo-function, and restores those register values after executing the pseudo-function. This means that the embedded instructions can use volatile registers without interfering with other code the compiler generates. Paragraph 5.2 gives additional information about register usage.

NOTES

The compiler cannot verify that the *instruction* value consists of valid PowerPC assembly-language instructions. Using the **__asmd( )** pseudo-function with any other instruction value can cause incorrect code.

The compiler loads *values* values into registers according to the parameter-passing conventions paragraph 5.4 explains.

The compiler does not know which assembly-language instructions you use in the **__asmd( )** pseudo-function, so the compiler cannot protect against any error that these instructions may cause by modifying registers.

Example:

```
{
        double fpscr;
        fpscr = __asmd("mffs f1\n")
}
```

In this example, the **`mffs`** instruction moves the contents of register FPSCR into **`f1`**. The pseudo-function returns the contents of **`f1`**, assigning them to the C variable **`fpscr`**.

CHAPTER 4
COMPILER OPTIMIZATIONS

This section explains considerations for optimization and the kinds of optimization that the compiler can perform.

4.1 CONSIDERATIONS FOR OPTIMIZATION

Previous chapters explain how to have the compiler apply many kinds of optimization to your source code, and how to specify the degree of these optimizations. Before you apply optimization, however, you should note these implications:

1. Compilation takes more time when it includes optimization. The higher the optimization level, the more analyses the compiler must perform, so the longer it takes to compile your program.
2. If different compilations with optimizations enabled cause your program to produce different results, a source program probably is incorrect.
3. Optimization can interfere with a source-level debugger. If you plan to use the Motorola Embedded Debugger (MEDB) to debug your code, Motorola strongly recommends that that you compile with the **`-g`** (include debugging) option and accept the default optimization level (**`-O1`**).

Also note that optimization usually gives better results when you submit more of the program to the compiler at one time. For example, assume that the source program loads a variable both before and after calling a subroutine. Also assume that the subroutine calls several other subroutines. If you submit only the main routine, the compiler must allow the subroutines to modify the variable. This means that the compiler must retain both load instructions.

But if you submit all the routines at the same time, the compiler can determine whether the subroutines can modify the variable. If they cannot, the compiler can decrease execution time by storing the variable value in a register, eliminating the second load instruction.

However, the drawbacks of a single, large compile invocation is that compilation takes significantly more time.

Paragraph 4.2 explains each of the possible MECC optimizations.

4.2 OPTIMIZATION TYPES

Paragraphs 4.2.1 through 4.2.17 explain the different types of optimizations that the compiler can perform. In some cases, a specific control variable governs the degree of one kind of optimization. In other cases, a control variable governs the degree of several kinds of optimization.

Not all these optimizations reduce program execution time by the same amount. But in many situations, an optimization that produces a marginal improvement itself enables other optimizations to make dramatic improvements.

4.2.1 Alias Analysis

Alias analysis is checking whether source code includes multiple references to the same memory object. Although not an optimization itself, alias analysis gathers information necessary for actual optimizations.

Alias analysis reveals whether two memory references cannot refer to the same memory object, in any possible execution. Such non-interfering references are opportunities for optimization.

At the same time, alias analysis determines whether there is any possible execution in which two references do pertain to the same memory object. Such interfering references, or *aliases*, limit the possible optimizations.

The compiler performs alias analysis based on:

- declarations of all variables that appear in the memory references.
- constant subscripts of non-pointer array references.
- pointer values within procedures.
- pointer values between procedures.

The value of the **-O** control option determines the number of these bases for alias analysis.

4.2.2 Call Modification Analysis

Call modification analysis is determining which objects can be modified by calling subroutines. Although not an optimization itself, call modification analysis gathers information necessary for actual optimizations.

Objects that cannot be modified because of a subroutine call may be opportunities for optimization. Objects that *can* be modified because of a subroutine call limit the possible optimizations.

For example, a subroutine can alter an argument passed by reference, so the argument is modifiable. Similarly, a subroutine can alter a C global variable, so the variable is modifiable.

The compiler performs call modification analysis inside procedures or between procedures, according to the value of the **-O** control option.

4.2.3 Eliminating Common Subexpressions

A common subexpression is an expression recomputation, even though there have been no value changes in the variables that make up the expression. Common subexpressions are unnecessary, so eliminating them speeds program execution.

For example, suppose that Line 2 of a block is:

```
r = 8*i/j;
```

that Line 6 is:

```
t = 50 + (8*i/j);
```

and that Line 11 is:

```
w = 8*i/j;
```

Lines 6 and 11 contain the common subexpression **8*i/j**. If the block has no reassignments of the **i** or **j** values between Lines 2 and 11, the compiler can substitute variable **r** for the subexpression **8*i/j** in lines 6 and 11. This eliminates two recalculations of the subexpression.

The value of the **-O** control option determines the level of eliminating common subexpressions.

4.2.4 Eliminating Dead Code

Dead code is source code that never will be executed, or for which results never will be used. Eliminating dead code reduces program size. (Even if none of the original source code is dead, some of the code can become dead after the compiler applies other optimizations.)

The value of the **-O** control option determines the level of dead-code elimination.

4.2.5 Hoisting Code out of Loops

Hoisting code out of loops is moving invariant computations or store instructions out of loops, to points outside. For example, if a while loop begins with the line:

```
while ( dewpt >= avtemp+10 )
```

the processor must compute the expression **avtemp+10** for each execution of the loop. But if nothing in the loop changes the value of **avtemp**, the compiler can move that calculation outside the loop, as in:

```
t = avtemp+10
while ( dewpt >= t )
```

This means that the processor calculates **avtemp+10** only once.

The value of the **-O** control option determines the level of hoisting code out of loops.

4.2.6 Strength Reduction

Strength reduction is substituting expressions that use computationally less-expensive operators (such as **+** and **-**) for expressions that have more powerful, time-consuming operators (such as ***** and **/**). For example, suppose that **a** and **b** are integers, and that a source-code line is:

```
a = b * 9;
```

The compiler recognizes that multiplication by 8 would give the same result as a three-place left shift. Addition of another **b** value would make the result the same as the original multiplication by 9. Accordingly, the compiler optimizes the assignment as:

```
a = (b << 3) + b;
```

Even together, the left-shift and addition operators are less expensive than the multiplication operator, so the replacement assignment line executes more quickly than the original.

The value of the **-O** control option determines the level of strength reduction.

4.2.7 Copy Propagation

Copy propagation is using copy **b** as much as possible after a value assignment of the form **b = a**, instead of using **a** again.

The compiler performs copy propagation within blocks (locally) or among blocks (globally). The value of the **-O** control option determines the level of this optimization.

4.2.8 Constant Propagation

Constant propagation is calculating at compile time the expressions that involve constant operands. Then, the compiler uses the constant values instead of variables (that is, propagates the constant values) where no intervening reassignments would affect the variables. For example, suppose that **a** and **b** are integers, and that a source-code line assigns a constant value to **a**:

```
a = 2;
```

Suppose that the value of **a** does not change before this subsequent line:

```
b = a + 25;
```

Knowing that **a** still has a constant value at this point, the compiler substitutes the constant for the variable, speeding execution:

```
b = 2 + 25;
```

(This optimization also enables a subsequent constant-folding optimization that would make the assignment **`b = 27;`**, further reducing program execution time.) The compiler performs constant propagation within blocks (locally) or among blocks (globally). The value of the **`-O`** control option determines the level of this optimization.

4.2.9 Forward Code Motion

Forward code motion is moving store instructions or invariant calculations from the body of a loop to a point after the loop. The idea of forward code motion is similar to that of hoisting code out of loops. But forward code motion pertains to situations in which the store or use of calculations can take place after the loop body; this difference can be significant due to the compiler's internal processing. Forward code motion is particularly important for store instructions.

For example, suppose that a while loop contains an assignment for variable **a**: the unoptimized program must carry out a store instruction each time it executes the loop:

```
while ( dewpt >= avtemp+10 )
{
  a = a + 10;
  .
  .
}
```

The optimizer establishes the temporary register ***r***, giving it the value of **a** before the loop begins. Within the loop, register ***r*** receives the incremented value. After the loop ends, **a** receives the final ***r*** value. This optimization reduces the previous multiple store instructions to just one.

```
r = a;
while ( dewpt >= avtemp+10 )
{
  r = r + 10;
  .
  .
}
a = r;
```

The compiler performs forward code motion for loops that do not have conditional code flow or for all loops. The value of the **-O** control option determines the level of this optimization.

4.2.10 Control Flow Optimization

Control flow optimization includes eliminating unreachable code, collapsing GOTOs that transfer to other GOTOs, merging adjacent basic blocks (where possible), and simplifying branches. For example, in the source code, instruction A jumps flow ahead to instruction B, but B is a jump instruction to C. As part of control flow optimization, the compiler rewrites instruction A as a jump to C and eliminates instruction B.

The value of the **-O** control option determines whether the compiler performs this optimization.

4.2.11 Loop Unrolling

Loop unrolling is changing the code of loops, to reduce the number of loop executions. This means that the system tests fewer conditions during program execution, so the program runs faster. Loop unrolling also increases the potential for other optimizations, such as schedule instruction improvements.

Complete unrolling would eliminate a loop by making its code linear. Although this ultimate goal rarely is possible, it often is possible for the optimizer to reduce the number of loop executions.

For example, this unoptimized loop executes 10 times:

```
for ( i = 1; i <= 10; i++ )
{
  a[i] = a[i] * b + 10
}
```

This unrolled loop carries out the same value assignments, but executes only five times:

```
for ( i = 1; i <= 5; i = i + 2)
{
  a[i] = a[i] * b + 10;
  a[i+1] = a[i+1] * b + 10;
}
```

The unrolled loop may lead to further time saving if the processor can execute store instructions in parallel. Such a processor can interleave the loads and stores of **`a[i]`** and **`a[i+1]`** with the actual calculations. Note, however, that loop unrolling increases code size.

The value of control variable **`unroll`** determines whether the compiler performs loop unrolling under automatic control or a specific number of times. The default value for this optimization is *not enabled*. (You can set the **`unroll`** value indirectly by specifying the value of the **`-O`** control option. The **`-O`** control option value 0 or 1 assigns the value 0 to **`unroll`**; other **`-O`** control option values assign the value 1 to **`unroll`**.)

4.2.12 Register Allocation

Register allocation is using registers to hold as many operand values as possible. Instructions involving register operands take much less time than instructions involving operands in memory. So an important way the compiler can speed program execution is to reassign specific memory values to the fast registers of the target processor.

Beyond the few values that *must* be allocated to registers (such as the first few parameters for a subroutine call), values that might well be held in registers are:

- Intermediate values involved in the language's expression evaluation or statement execution. Examples are subexpressions of the evaluated expressions and quantities involved in addressing expressions.
- Scalar C variables: variables that are not arrays, array elements, structures, or structure elements. Such scalars must have automatic extent and their addresses must not be taken with the **&** operator.

As the number of available registers rarely equals the values that could be stored in registers, the compiler attempts to find the greatest optimization benefit. One approach is to use registers to hold the most frequently used variables. Where possible, the compiler even makes a register the place where a variable lives, instead of storing the variable in memory.

Another part of register allocation is considering the order of computations: this order also can affect program execution speed, so the compiler must decide whether to change the original order of routines.

The compiler performs register allocation inside procedures, between procedures without reordering routines, or between procedures while reordering routines. The value of the **-O** control option determines the level of this optimization.

4.2.13 Instruction Scheduling

Instruction scheduling is reordering machine instructions to exploit the characteristics of the target processor. If, for example, the target processor lets certain instructions execute in parallel, the compiler reorders instructions appropriately, reducing program execution time.

The compiler performs instruction scheduling during compiler pass one or during compiler passes one and two. The value of control variable **sched** determines the level of this optimization. Alternatively, the value of the **-O** control option can determine the level of this optimization.

4.2.14 Eliminating Loop Induction Variables

Loop induction variables are secondary or tertiary loop-control variables that have some consistent relationship with the primary loop-control variable, no matter how many times the processor executes the loop. The compiler uses the primary loop-control variable to devise new expressions that substitute for the loop induction variables. This lets the compiler eliminate the loop induction variables, reducing program execution time.

The value of the **-O** control option determines whether the compiler performs this optimization.

4.2.15 Global Instruction Movement

Global instruction movement is moving instructions across basic blocks, to improve instruction scheduling. Note that a basic block is a sequence of instructions with these qualities: execution must begin with the first instruction of the block, all contained instructions must be executed, and execution ends with that last instruction of the block (that is, there cannot be any branches out of the block).

The compiler performs global instruction movement for basic blocks executed non-conditionally or for all possible blocks. The value of control variable **gim** determines the level of this optimization.

4.2.16 Inlining Functions

Inlining a function is moving its instructions to the calling routine. If it is possible to inline a particular function, the compiler eliminates the overhead of calling the function and returning from the function.

Two MECC control variables give you control over this optimization: **`inline`** lets you list the functions the compiler should consider for inlining, and **`inllev`** determines the size and complexity of functions the compiler should inline. Alternatively, the value of the **`-O`** control option can determine the level of this optimization, provided that you have supplied a value to **`inline`**.

4.2.17 Multiple Return Points

A return point is a sequence of instructions that returns program flow to the calling routine. Typically, a routine that has multiple return points executes faster than an equivalent routine that has only one return point, even though the multiple-return-point routine code is larger. A programmer can write C code that includes multiple return points; an optimizing compiler can add return points as it compiles the code. Additionally, some other optimizations may create multiple return points from a single source-code return statement, further enhancing performance.

The value of control variable **`retpts`** determines whether the compiler may create multiple return points.

4.3 SETJMP AND LONGJMP FUNCTIONS

The functions **`setjmp`** and **`longjmp`** bear on appropriate levels of optimization. Processing for a **`setjmp`** call includes storing register values in a buffer. Processing the return from **`longjmp`** (back to the **`setjmp`** call) retrieves those values from the buffer and restores them to the registers.

If **`setjmp`** returns with a nonzero value, static variables have their proper values as of the time **`longjmp`** was called. Automatic (stack) variables local to the function containing the **`setjmp`** call have their correct values in ANSI C only if the variables have been declared volatile or if the values were not changed between the original call to **`setjmp`** and the corresponding **`longjmp`** call.

In other words, if a program modifies the value of a register-resident variable between the **`setjmp`** call and the **`longjmp`** call, the **`setjmp`**/**`longjmp`** mechanism restores the register set to the values at the time of the original **`setjmp`** call.

The higher the optimization level, the greater the likelihood that automatic variables have been kept in registers, rather than in memory. Therefore:

> If, between **`setjmp`** and **`longjmp`** calls, program code modifies any local variables that have automatic scope, your program **must not** subsequently use such variables in any way that affects program correctness.

CHAPTER 5
EMBEDDED APPLICATION BINARY INTERFACE

To work smoothly with the output of the compiler, your assembly-language functions should comply with the PowerPC™ Embedded Application Binary Interface (EABI) standard.

This section explains the most important EABI conventions, with regard to embedded applications. For a discussion of all the conventions, consult the EABI standard itself.

5.1 DATA FORMATS

Data types in both your assembly-language functions and your C-language functions should conform to the C scalar types. Table 5-1 lists these data types, along with the related assembler pseudo operations.

Table 5-1. C Scalar Data Types

Scalar Type	Storage Element	Alignment	PowerPC Format
signed char	byte	1 byte	signed byte
char, unsigned char	byte	1 byte	unsigned byte
short, signed short	halfword	2 bytes	signed halfword
unsigned short	halfword	2 bytes	unsigned halfword
int, signed int, long, signed long, enum	word	4 bytes	signed word
unsigned int, unsigned long	word	4 bytes	unsigned word
long long, signed long long	doubleword	8 bytes	signed doubleword
unsigned long, long	doubleword	8 bytes	unsigned doubleword
pointer to anything(1)	word	4 bytes	unsigned word
float	word	4 bytes	IEEE 32-bit float
double	doubleword	8 bytes	IEEE 64-bit float
long double(2)	quadword	16 bytes	quadword (IEEE 128-bit float)

(1) That is, a pointer to an object of any type.

(2) The long double format is 16-byte IEEE double extended precision with a sign bit, 15-bit exponent (bias of -16383), and a 112-bit fraction.

These examples show equivalent C and assembly-language data definitions:

```
long i = 3;          i:  .long   3
double j;            j:  .double 0.0
```

5.2 REGISTER USAGE CONVENTIONS

The PowerPC architecture has 32 general-purpose registers, GPR0 through GPR31, which the assembler refers to as **r0** through **r31**. Each GPR is 32 bits wide; Table 5-2 shows the usage conventions for these registers.

Note that registers **r0**, and **r3** through **r12** are *volatile*: a called function can modify the contents of volatile registers without restoring the values. Thus, if function **a_one** calls function **b_two**, **b_two** could modify these volatile registers and return without restoring their values. Consequently, **a_one** must save its own volatile values before calling **b_two**, then restore these volatile values after the return from **b_two**.

Registers **r13** through **r31** are *nonvolatile*. A called function must save the contents of nonvolatile registers before modifying them. The called function must restore the contents of nonvolatile registers before returning to the calling function.

Table 5-2. General Purpose Register (GPR) Conventions

Register	Status	Use
r0	volatile	For function prologs, epilogs, and other language-specific uses.
r1	dedicated	Stack pointer (SP): always points to the lowest allocated valid stack frame, growing toward low addresses. The word **r1** points to contains the address of the previously allocated stack frames. The called function can decrement the stack frame.
r2	dedicated	Contains the base of the ELF segment **.sdata2**, if the object file has that segment. (The base is a kind of central address for the segment: every byte of the segment is within a signed, 16-bit offset of the base.) User routines do not modify **r2**.
r3, r4	volatile	For parameter passing and return values.
r5 — r10	volatile	For parameter passing.
r11, r12	volatile	For function prologs, epilogs, and other language-specific uses.
r13	dedicated	Contains the base of the ELF segment **.sdata**, if the object file has that segment. (The base is a kind of central address for the segment: every byte of the segment is within a signed, 16-bit offset of the base.) User routines do not modify **r13**.
r14 — r31	nonvolatile	For local calculation.

The PowerPC architecture defines 32 floating-point registers, FPR0 through FPR31, which the assembler refers to as **`f0`** through **`f31`**. Each GPR is 64 bits wide; Table 5-3 shows the usage conventions for these registers. (Note that registers **`f0`** through **`f13`** are *volatile*.)

Table 5-3. Floating Point Register (FPR) Conventions

Register	Status	Use
`f0`	volatile	For language-specific uses.
`f1`	volatile	For parameter passing and return values.
`f2 — f8`	volatile	For parameter passing.
`f9 — f13`	volatile	For any uses.
`f14 — f31`	nonvolatile	For any uses.

Note these additional conventions:

- Any routine that calls a variable-argument-list function must define the value of bit 6 of the condition register (CR) immediately before calling the function. (Paragraph 5.5 explains more about such functions.)
- The link register (LR) contains the address to which a called function normally returns. Functions return by means of a **`blr`** instruction.

5.3 STACK FRAMES

Each function call adds a stack frame to the run-time stack storage area. Figure 5-1 shows the layout of a stack frame.

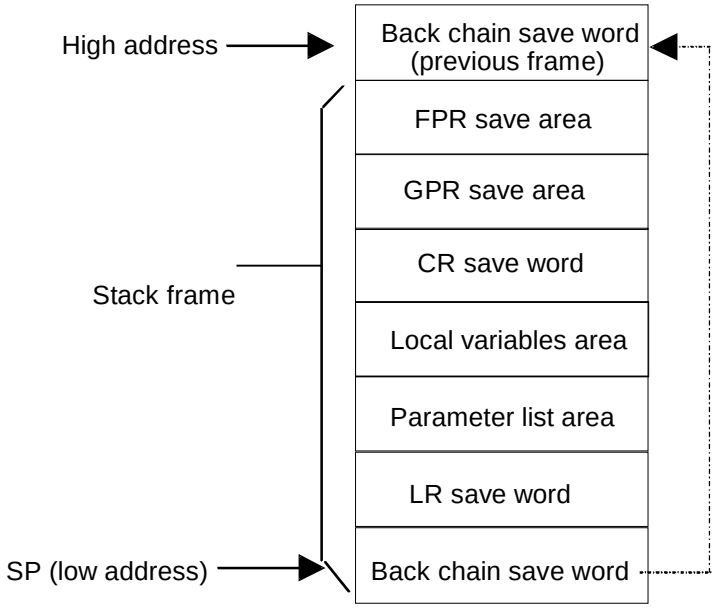

Figure 5-1. EABI Stack Frame Layout

Source code must comply with these stack frame requirements:

1. The stack frame alignment is on an eight-byte boundary. The stack grows downward, that is, toward lower addresses. If stack-frame padding is necessary to maintain an 8-byte alignment, the padding must go into the local variables area.
2. The stack pointer (SP) points to the first word of the lowest stack frame: the *back chain word*. The back chain word points to the low address (beginning) of the previous stack frame, that is, to the stack frame of the calling function. (The first stack frame of the stack is an exception: its back chain word has a zero value.)
3. If a called function needs another stack frame, the function must decrement the SP value before calling another function or storing values in a stack-frame area. Before executing the **`blr`** instruction that returns program flow to the calling function, the called function must restore the original SP value.

4. To decrement the SP value and update the back chain value, code must use one of the *Store Word with Update* instructions. The code must use a single instruction to restore the SP value. This makes sure that the SP always points to the beginning of a linked list of stack frames.
5. Except in the *Store with Update* instructions, offsets from the SP must not be negative.
6. The calling routine allocates the parameter list area, making certain that this area is large enough to contain all the called arguments.
7. If a function changes the value of any nonvolatile floating-point register, it first must save the function-start values of that register and all higher-numbered floating-point registers. The function stores these values in consecutive double words in the FPR save area, so that the value of **`f31`** is in the double word immediately below the back chain word of the previous stack frame. (The nonvolatile floating-point registers are **`f14`** through **`f31`**.) For example, in order to change the value in **`f20`**, the function first must use the FPR save area to store the values that registers **`f20`** through **`f31`** had when the function began.
8. If a function changes the value of any nonvolatile general-purpose register, it first must save the function-start values of that register and all higher-numbered general-purpose registers. The function stores these values in consecutive words in the GPR save area, so that the value of **`r31`** is in the word immediately below the FPR save area. (The nonvolatile floating-point registers are **`r14`** through **`r31`**.) For example, in order to change the value in **`r24`**, the function first must use the GPR save area to store the values that registers **`r24`** through **`r31`** had when the function began.
9. If a function changes the value of any nonvolatile field of the condition register (CR), it first must save the function-start values of all CR fields. The function stores these values in the CR save area.
10. The minimum stack frame consists of the back chain word and the LR save word. (The standard calling sequence does not define a maximum stack-frame size.) The sizes of other stack-frame areas depend on the assembly code algorithm. If a function will not use other areas, it does not have to allocate space for them.
11. The LR save area is for storing the link register (LR) value, in case the function calls another function. The called function stores the LR value in the LR save area of the calling function's stack frame. For example, function A calls function B; when B calls function C, B saves the LR value in A's stack frame.

If a function does not call any other functions and does not require any stack-frame areas, the function does not have to create a stack frame.

5.4 PARAMETER PASSING

Using registers to pass parameter values, instead of using stack memory, makes the program execute faster. But the finite number of registers and EABI constraints on types of values allowed in registers limit this practice.

The PowerPC EABI lets user registers **r3** through **r10** pass as many as eight non-floating-point arguments; it lets user registers **f1** through **f8** pass as many as eight floating-point arguments. (If a function's arguments do not need all these registers, the extra registers have undefined values when function execution begins.)

Only if all the arguments do not fit into the registers should code allocate stack frame space for argument storage. Even then, the code should allocate only enough space to hold the extra arguments. Code must assign arguments to registers per the algorithm of paragraph 5.4.1. Paragraph 5.4.2 shows an example of such argument passing.

5.4.1 Argument Passing Algorithm

This paragraph shows an appropriate algorithm for passing arguments. Explanatory comments follow this algorithm.

INITIALIZE: Set **fr** to 1, **gr** to 3, and **starg** to the low address of the stack frame's parameter list area. (*Exception*: if the function to be called returns a long double, or a struct or union longer than 8 bytes, set **gr** to 4.)

SCAN: If no more arguments, terminate.

Else, if argument is a 4-byte or smaller integer, is a pointer, or has been converted to being passed as a pointer (that is, is a struct, union, or long double):

Sign- or zero-extend argument value to 32 bits.

If **gr** < 11, pass value in GPR **gr**, add 1 to **gr**, and go to SCAN. *Otherwise*, go to LIST_AREA and treat argument as a 4-byte long, 4-byte aligned value.

Else, if argument is an 8-byte integer (such as non-ANSI C long long):

If **gr** < 10, add 0 or 1 to **gr** to make it an odd number, pass value's lower-addressed word in GPR **gr**, pass value's other word in GPR **gr+1**, add 2 to **gr**, and go to SCAN. *Otherwise*, go to LIST_AREA and treat argument as an 8-byte long, 8-byte aligned value.

Else, if argument is a 4-byte or 8-byte float:

Convert value to 8-byte float.

If **`fr`** < 9, pass value in FPR **`fr`**, add 1 to **`fr`**, and go to SCAN. *Otherwise*, go to LIST_AREA and treat argument as an 8-byte long, 8-byte aligned value.

LIST_AREA: *Else*, if argument is not handled above:

Round up **`starg`**, as needed, so that **`starg`** is a multiple of the argument's alignment.

Copy the value, lowest to highest byte, to the location to which **`starg`** points.

Add the argument's size to **`starg`** and go to SCAN.

Comment 1. A simple argument has a four-byte size and alignment. Code sign-extends a signed simple-argument value shorter than 32 bits. Code zero-extends an unsigned simple-argument value shorter than 32 bits. A simple argument is one of these types:

- A simple integer (char, short, int, long, or enum) no more than 32 bits wide.
- A pointer to an object of any type.
- A pointer to a structure or union.

Comment 2. Executable code in the called function must treat a structure or union argument as a pointer to that structure or union object (or to a copy of the object). In a call-by-reference environment, the pointer points to the object itself; in a call-by-value environment (such as C), the pointer points to a copy of the object.

Comment 3. Float and double arguments have eight-byte size and alignment; float arguments are converted to double representation when passed in FPRs.

5.4.2. Argument Passing Example

Suppose the system runs this code:

```
typedef struct
{
   int a, b;
   double dd;
} sparm;
sparm s, t;
int c, d, e, f, g, h, i;
double ff, gg, hh, ii, jj, kk, ll, mm, nn;

x = func(c, ff, d, gg, e, hh, f, ii, g, jj, h, kk, i, ll, s, mm,
t, nn);
```

Calling the function as shown assigns these values to registers and the stack frame:

- GPRs:

  r3: c **r6**: f **r9**: i

  r4: d **r7**: g **r10**: pointer to **s**

  r5: e **r8**: h

- FPRs:

  f1: ff **f4**: ii **f7**: ll

  f2: gg **f5**: jj **f8**: mm

  f3: hh **f6**: kk

- Stack frame offsets:

  08: pointer to **t**

  0c: nn (low word)

  10: nn (high word)

5.5 VARIABLE ARGUMENTS

In C, functions may have variable-length arguments, or *varargs*. The ellipsis (...) in the function prototype indicates a vararg. The **`printf`** function, for example, can take a vararg list.

The PowerPC EABI specifies that some arguments be passed in registers and that others be passed on the stack. Accordingly, a vararg function must call the routine **`_va_arg`** to find the locations and types of the arguments in the list.

Even if you do not need to write vararg routines, you should know how to write routines that call **`printf`** or other vagarg routines. If such a calling routine passes any arguments in FPRs, the calling routine must assign the value *1* to CR bit 6, via this instruction:

```
creqv    6, 6, 6     #set CR bit 6
```

A function that takes a vararg can test CR bit 6: the value *1* tells the function that arguments are in FPRs. Accordingly, the function must perform eight doubleword saves.

For a routine that calls a vararg routine but does *not* pass any arguments in FPRs, Motorola recommends that the calling routine assign the value *0* to CR bit 6, via this instruction:

```
crxor    6, 6, 6     #clear CR bit 6
```

If the vararg function tests CR bit 6, the value *0* tells the function that no arguments are in FPRs. This does away with the need for eight doubleword saves.

5.6 RETURN VALUES

A function returns a float or double value in **`f1`**, rounding a float. A function returns an int, long, enum, short, char, or pointer value in **`r3`**. The function rounds, sign-extends, or zero-extends an **`r3`** return value as appropriate.

A function returns a long long, or a union or structure of eight or fewer bytes, in **`r3`** and **`r4`**. The function returns this value as if it had been read from memory: the lower-addressed word in **`r3`**. Values in bytes of **`r3`** and **`r4`** after the end of the return value are undefined.

For a return long double, or a return structure or union of more than eight bytes, the function uses a storage buffer that the calling function allocates. The calling function passes the address of this buffer as a hidden argument in **r3**, as if it were the first argument. (For the argument-passing algorithm of paragraph 5.4.1, this means that **gr** must be initialized to *4*, instead of *3*. As a result, the registers used for argument passing are **r4** through **r10**.)

5.7 FUNCTION PROLOGS AND EPILOGS

Each function typically consists of three code sections: the prolog, the main body, then the epilog. The code of the main body carries out the tasks of the function; previous paragraphs pertained to the main body. This paragraph covers the prolog and epilog:

- The prolog sets up registers and performs other housekeeping tasks required before the main body can execute. A prolog establishes a stack frame, if necessary, and may save any nonvolatile registers that it uses.
- The epilog restores registers the prolog saved, restores the previous stack frame, and returns to the calling function.

5.7.1 Prolog and Epilog Rules

The EABI does not dictate specific code sequences for prologs or epilogs. But your prologs and epilogs must follow these rules (which permit call chain backtracking):

1. Before a function calls any other function, the calling function must establish its own stack frame. The size of this stack frame must be a multiple of eight bytes. The calling function must save the link-register (LR) value (at the time execution of the calling function began) in the LR save word of the previous stack frame. (Assume, for example, that function A calls function B, which calls function C. As part of calling C, B establishes its own stack frame and saves the LR value in A's stack frame.)
2. If a function establishes a stack frame, the function must use a *Store Word with Update* instruction to update atomically the back chain word of the stack frame with the SP.
    a. For a stack frame no larger than 32 kilobytes, the prolog should use the **stwu** instruction, with an appropriate negative displacement.
    b. For a larger stack frame, the prolog must use the **addis** and **addi** instructions or the **ori** instruction to compute the frame size. Then the prolog must load a volatile register with the two's complement of the frame size and issue a **stwux** instruction.
3. When a function deallocates its stack frame, it must do so atomically using a single instruction: either by loading the back chain value into the SP (**r1**) or by incrementing the stack pointer (by the amount of the previous stack-pointer decrement).

4. If a function saves a nonvolatile GPR, the function also should save all higher-numbered GPRs. If a function saves a nonvolatile FPR, the function also should save all higher-numbered FPRs.

5.7.2 System Subroutines

A prolog may use in-line code to save volatile GPRs or FPRs; similarly, an epilog may use in-line code to restore such values. But if a function must save and restore many register values, it may be more efficient to call one of these subroutines:

- **savefpr_l.o routines.** These routines save FPR values in the FPR save area. These routines also save **`r0`** (which contains the function-entry LR value).

```
   .extern _savefpr_l4_l
   .extern _savefpr_l5_l
   .extern _savefpr_l6_l
   .extern _savefpr_l7_l
   .extern _savefpr_l8_l
   .extern _savefpr_l9_l
   .extern _savefpr_20_l
   .extern _savefpr_21_l
   .extern _savefpr_22_l
   .extern _savefpr_23_l
   .extern _savefpr_24_l
   .extern _savefpr_25_l
   .extern _savefpr_26_l
   .extern _savefpr_27_l
   .extern _savefpr_28_l
   .extern _savefpr_29_l
   .extern _savefpr_30_l
   .extern _savefpr_31_l

   .sect  .text
   .long  0x00400000    #tag
_savefpr_l4_l:         stfd f14, -144(r11)
_savefpr_l5_l:         stfd f15, -136(r11)
_savefpr_l6_l:         stfd f16, -128(r11)
_savefpr_l7_l:         stfd f17, -120(r11)
_savefpr_l8_l:         stfd f18, -112(r11)
_savefpr_l9_l:         stfd f19, -104(r11)
_savefpr_20_l:         stfd f20, -96(r11)
_savefpr_21_l:         stfd f21, -88(r11)
_savefpr_22_l:         stfd f22, -80(r11)
_savefpr_23_l:         stfd f23, -72(r11)
_savefpr_24_l:         stfd f24, -64(r11)
_savefpr_25_l:         stfd f25, -56(r11)
_savefpr_26_l:         stfd f26, -48(r11)
 _
```

```
 savefpr_27_l:        stfd f27, -40(r11)
_savefpr_28_l:         stfd f28, -32(r11)
_savefpr_29_l:         stfd f29, -24(r11)
_savefpr_30_l:         stfd f30, -16(r11)
_savefpr_31_l:         stfd f31, -8(r11)
  stw r0,  4(r11)
  blr
```

- **restfpr_l.o routines**. These routines restore FPR values, restore the LR and the stack frame pointer of the calling routine, and return to the calling routine.

```
   .extern _restfpr_l4_l
   .extern _restfpr_l5_l
   .extern _restfpr_l6_l
   .extern _restfpr_l7_l
   .extern _restfpr_l8_l
   .extern _restfpr_l9_l
   .extern _restfpr_20_l
   .extern _restfpr_21_l
   .extern _restfpr_22_l
   .extern _restfpr_23_l
   .extern _restfpr_24_l
   .extern _restfpr_25_l
   .extern _restfpr_26_l
   .extern _restfpr_27_l
   .extern _restfpr_28_l
   .extern _restfpr_29_l
   .extern _restfpr_30_l
   .extern _restfpr_31_l

   .sect  .text
   .long  0x00600000    #tag
_restfpr_l4_l:         lfd  f14, -144(r11)
_restfpr_l5_l:         lfd  f15, -136(r11)
_restfpr_l6_l:         lfd  f16, -128(r11)
_restfpr_l7_l:         lfd  f17, -120(r11)
_restfpr_l8_l:         lfd  f18, -112(r11)
_restfpr_l9_l:         lfd  f19, -104(r11)
_restfpr_20_l:         lfd  f20, -96(r11)
_restfpr_21_l:         lfd  f21, -88(r11)
_restfpr_22_l:         lfd  f22, -80(r11)
_restfpr_23_l:         lfd  f23, -72(r11)
```

```
_restfpr_24_l:        lfd  f24, -64(r11)
_restfpr_25_l:        lfd  f25, -56(r11)
_restfpr_26_l:        lfd  f26, -48(r11)
_restfpr_27_l:        lfd  f27, -40(r11)
_restfpr_28_l:        lfd  f28, -32(r11)
_restfpr_29_l:        lfd  f29, -24(r11)
_restfpr_30_l:        lfd  f30, -16(r11)
_restfpr_31_l:        lwz  r0, 4(r11)
  lfd f31,  -8(r11)
  mtlr r0
  ori r1, r11, 0
  blr
```

- **savegpr_l.o routines.** These routines save GPR values in the GPR save area. These routines also save the LR value previously stored in **r0**. You may use these routines whether or not FPR values already have been saved.

```
   .extern _savegpr_14_l
   .extern _savegpr_15_l
   .extern _savegpr_16_l
   .extern _savegpr_17_l
   .extern _savegpr_18_l
   .extern _savegpr_19_l
   .extern _savegpr_20_l
   .extern _savegpr_21_l
   .extern _savegpr_22_l
   .extern _savegpr_23_l
   .extern _savegpr_24_l
   .extern _savegpr_25_l
   .extern _savegpr_26_l
   .extern _savegpr_27_l
   .extern _savegpr_28_l
   .extern _savegpr_29_l
   .extern _savegpr_30_l
   .extern _savegpr_31_l

   .sect  .text
   .long  0x00400000    #tag
_savegpr_l4_l:       stw   r14, -72(r11)
_savegpr_l5_l:       stw   r15, -68(r11)
_savegpr_l6_l:       stw   r16, -64(r11)
```

```
_savegpr_l7_l:       stw   r17, -60(r11)
_savegpr_l8_l:       stw   r18, -56(r11)
_savegpr_l9_l:       stw   r19, -52(r11)
_savegpr_20_l:       stw   r20, -48(r11)
_savegpr_21_l:       stw   r21, -44(r11)
_savegpr_22_l:       stw   r22, -40(r11)
_savegpr_23_l:       stw   r23, -36(r11)
_savegpr_24_l:       stw   r24, -32(r11)
_savegpr_25_l:       stw   r25, -28(r11)
_savegpr_26_l:       stw   r26, -24(r11)
_savegpr_27_l:       stw   r27, -20(r11)
_savegpr_28_l:       stw   r28, -16(r11)
_savegpr_29_l:       stw   r29, -12(r11)
_savegpr_30_l:       stw   r30, -8(r11)
_savegpr_31_l:       stw   r31, -4(r11)
   stw  r0,  4(r11)
   blr
```

- **savegpr.o routines.** These routines also save GPR values in the GPR save area, but only if FPR values already have been saved. These routines also save the LR value previously stored in **r0**.

```
.extern _savegpr_14
.extern _savegpr_15
.extern _savegpr_16
.extern _savegpr_17
.extern _savegpr_18
.extern _savegpr_19
.extern _savegpr_20
.extern _savegpr_21
.extern _savegpr_22
.extern _savegpr_23
.extern _savegpr_24
.extern _savegpr_25
.extern _savegpr_26
.extern _savegpr_27
.extern _savegpr_28
.extern _savegpr_29
.extern _savegpr_30
.extern _savegpr_31
```

```
   .sect  .text
   .long  0x00400000    #tag
_savegpr_14:         stw   r14, -72(r11)
_savegpr_15:         stw   r15, -68(r11)
_savegpr_16:         stw   r16, -64(r11)
_savegpr_17:         stw   r17, -60(r11)
_savegpr_18:         stw   r18, -56(r11)
_savegpr_19:         stw   r19, -52(r11)
_savegpr_20:         stw   r20, -48(r11)
_savegpr_21:         stw   r21, -44(r11)
_savegpr_22:         stw   r22, -40(r11)
_savegpr_23:         stw   r23, -36(r11)
_savegpr_24:         stw   r24, -32(r11)
_savegpr_25:         stw   r25, -28(r11)
_savegpr_26:         stw   r26, -24(r11)
_savegpr_27:         stw   r27, -20(r11)
_savegpr_28:         stw   r28, -16(r11)
_savegpr_29:         stw   r29, -12(r11)
_savegpr_30:         stw   r30, -8(r11)
_savegpr_31:         stw   r31, -4(r11)
   blr
```

- **restgpr_l.o routines**. These routines restore GPR values, restore the LR and the stack frame pointer of the calling routine, and return to the calling routine. You may use these routines whether or not FPR values already have been restored.

```
   .extern _restgpr_14_l
   .extern _restgpr_15_l
   .extern _restgpr_16_l
   .extern _restgpr_17_l
   .extern _restgpr_18_l
   .extern _restgpr_19_l
   .extern _restgpr_20_l
   .extern _restgpr_21_l
   .extern _restgpr_22_l
   .extern _restgpr_23_l
   .extern _restgpr_24_l
   .extern _restgpr_25_l
   .extern _restgpr_26_l
   .extern _restgpr_27_l
   .extern _restgpr_28_l
   .extern _restgpr_29_l
   .extern _restgpr_30_l
   .extern _restgpr_31_l
```

```
   .sect  .text
   .long  0x00400000    #tag
_restgpr_l4_l:       lwz   r14, -72(r11)
_restgpr_l5_l:       lwz   r15, -68(r11)
_restgpr_l6_l:       lwz   r16, -64(r11)
_restgpr_l7_l:       lwz   r17, -60(r11)
_restgpr_l8_l:       lwz   r18, -56(r11)
_restgpr_l9_l:       lwz   r19, -52(r11)
_restgpr_20_l:       lwz   r20, -48(r11)
_restgpr_21_l:       lwz   r21, -44(r11)
_restgpr_22_l:       lwz   r22, -40(r11)
_restgpr_23_l:       lwz   r23, -36(r11)
_restgpr_24_l:       lwz   r24, -32(r11)
_restgpr_25_l:       lwz   r25, -28(r11)
_restgpr_26_l:       lwz   r26, -24(r11)
_restgpr_27_l:       lwz   r27, -20(r11)
_restgpr_28_l:       lwz   r28, -16(r11)
_restgpr_29_l:       lwz   r29, -12(r11)
_restgpr_30_l:       lwz   r30, -8(r11)
_restgpr_31_l:       lwz   r12, 4(r11)
   lwz  r31, -4(r11)
   mtlr r12
   ori r1, r11, 0
   blr
```

- **restgpr.o routines**. These routines restore GPR values, restore the LR and the stack frame pointer of the calling routine, and return to the calling routine, but only if the FPR values already have been restored.

```
   .extern _restgpr_14
   .extern _restgpr_15
   .extern _restgpr_16
   .extern _restgpr_17
   .extern _restgpr_18
   .extern _restgpr_19
   .extern _restgpr_20
   .extern _restgpr_21
   .extern _restgpr_22
   .extern _restgpr_23
   .extern _restgpr_24
   .extern _restgpr_25
```

```
    .extern _restgpr_26
    .extern _restgpr_27
    .extern _restgpr_28
    .extern _restgpr_29
    .extern _restgpr_30
    .extern _restgpr_31

    .sect  .text
    .long  0x00600000    #tag
_restgpr_14:         lwz   r14, -72(r11)
_restgpr_15:         lwz   r15, -68(r11)
_restgpr_16:         lwz   r16, -64(r11)
_restgpr_17:         lwz   r17, -60(r11)
_restgpr_18:         lwz   r18, -56(r11)
_restgpr_19:         lwz   r19, -52(r11)
_restgpr_20:         lwz   r20, -48(r11)
_restgpr_21:         lwz   r21, -44(r11)
_restgpr_22:         lwz   r22, -40(r11)
_restgpr_23:         lwz   r23, -36(r11)
_restgpr_24:         lwz   r24, -32(r11)
_restgpr_25:         lwz   r25, -28(r11)
_restgpr_26:         lwz   r26, -24(r11)
_restgpr_27:         lwz   r27, -20(r11)
_restgpr_28:         lwz   r28, -16(r11)
_restgpr_29:         lwz   r29, -12(r11)
_restgpr_30:         lwz   r30, -8(r11)
_restgpr_31:         lwz   r31, -4(r11)
    blr
```

An example of prolog and epilog code completes this section. (This example code is for a function that does not alter the nonvolatile fields of the CR or the FPSCR and that does no dynamic stack allocation.)

```
            .sect  .text
            .long  0x0000252C                #tag word
function:   ori    r11, r1, 0                #save end-of-frame
            stwu   r1, -length(r1)           #establish new frame
            mflr   r0                        #save LR in r0
            bl     _savefpr_14_1             #save LR, FPRs
            la     r11, -144(r11)            #set up for, and
            bl     _savegpr_14               #save GPRs
#save CR, if needed
...                                          #function main body
            addi   r11,r1, length-144        #restore GPRs
            bl     _restgpr_14
#restore CR, if needed
            addi   r11, r1, length           #restore FPRs, LR,
            bl     _restfpr_14_1             #r1, and return
```

5.8 INSTRUCTION-SET RESTRICTIONS

An application may use any PowerPC processor instruction, except for *Load/Store Multiple Word* instructions and *Load/Store String* instructions. Do not use any of these instructions in Little-Endian applications, as they may cause alignment exceptions. In Big-Endian applications, such an instruction usually is slower than a sequence of other instructions that has the same effect.

5.9 SMALL DATA AREAS

The EABI defines three small data areas (SDAs): data storage areas configured for very efficient storage or retrieval. Every address in an SDA is within a 16-bit, signed offset from the SDA base address. This means that each SDA can store as much as 64K bytes of data.

One of the SDAs always uses address 0 as its base address. The other two SDAs have specific registers, called *base registers*, that contain their base addresses.

The three SDAs are:

- ELF sections **.sdata** and **.sbss**; the base register is **r13**. The linker or loader determines the base address, assigning it to symbol **_SDA_BASE_**.
- ELF sections **.sdata2** and **.sbss2**; the base register is **r2**. The linker or loader determines the base address, assigning it to symbol **_SDA2_BASE_**.
- ELF sections **.PPC.EMB.sdata0** and **.PPC.EMB.sbss0**. The base address always is 0.

Your code can access data stored in SDAs via a single load or store instruction. Similarly, a single instruction can produce the address of data stored in an SDA. This can reduce the size of your code; this can increase the execution speed of your code.

For example, if *site3* is a four-byte variable, in section **.sdata**, these code lines load site3's value and address:

```
lwz  r5,sda(site3)(r13) # loads r5 with value of site3
addi r6,r13,sda(site3)  # loads r6 with address of site3
```

APPENDIX A
MECC ERROR MESSAGES

The compiler features an error-detection subsystem that issues remarks, warnings, and error messages. Table A-1 lists many of these messages, along with the probable causes and appropriate corrective actions. Messages not in this table should be self-explanatory.

Table A-1. MECC Error Messages

Message	Probable Cause	Corrective Action
argument to macro is empty	Missing argument in macro calling statement.	Supply the missing argument.
array is too large	Handling addresses for this array would exceed limits.	Shrink one or more array elements.
array of functions is not allowed	Array elements must have known, consistent size; functions have neither.	Convert to array of function pointers.
array of void is not allowed	Array elements must be typed; void is for untyped data.	Convert elements to some non-void data type.
bit field cannot contain all values of the enumerated type	Struct bit field too small for the enumerated type.	In the struct declaration, specify enough bits for the type.
cast between pointer-to-object and pointer-to-function	Code attempts to cast a pointer to a variable into a pointer to a function (or vice versa).	Do not use pointers to variables and pointers to functions interchangeably.
conversion of non-zero integer to pointer	Code assigns (or uses) an integer constant in a context where the compiler expected a pointer. The compiler converted the constant to a pointer.	Do not assign integer constant values to pointers.

Table A-1. MECC Error Messages (cont.)

Message	Probable Cause	Corrective Action
could not open source file <filename>.	File <filename> does not exist.	Create file <filename>.
	You do not have file permissions to open file <filename>.	Set appropriate permissions.
	An #include statement specified file <filename>, but <filename> is not in any directory of the -I (include) search list.	Make sure that the -I search list contains the directory in which file <filename> resides.
could not open temporary file <filename>.	Environment variable TMPDIR specifies a directory that does not exist.	Change TMPDIR or create the specified directory.
	You do not have sufficient file permissions for the directory that TMPDIR specifies.	Set appropriate permissions for the specified directory.
	The directory TMPDIR specifies is full.	Delete unneeded files from the directory.
declaration does not declare anything	Accidentally omitting the variable declarator list in a declaration, as in: `int;`	Remove the empty declaration.
definition for function <function> is missing	Code declares and references a non-external (static) function, but does not define the function.	Define the missing static function in the source file.
directive not allowed -- an #else has already appeared	#else or #elif in a conditional structure that already has an #else.	Remove the inappropriate #else or #elif, or change the first #else to an #elif.
division by zero	A constant expression that evaluates to zero was specified as a divisor.	Change the divisor or modify the expression to avoid division by zero.
enumerated type mixed with another type	Association of an enum-type variable to another variable that is not of the same enum type.	Use explicit type casts during enum assignments.

Table A-1. MECC Error Messages (cont.)

Message	Probable Cause	Corrective Action
expression must be a modifiable lvalue	Expression is not a modifiable lvalue, but context requires it.	Replace the expression with a complete lvalue not of type array, not cross-qualified, and not a struct or union.
expression must be an integral constant expression	Expression contains references to non-constant or non-integral data.	Replace the expression with an integral constant expression: one that evaluates to an integral-, character-, or enumeration-type constant.
expression must be an lvalue	Expression is not an lvalue, but context requires it.	Replace the expression with an lvalue: an expression that refers to an object in a way that permits examination and alteration of the object.
expression must be an lvalue or a function designator	Expression is not an lvalue or function name, but context requires one or the other.	Replace the expression with an lvalue or function name.
external/internal linkage conflict with previous declaration	Code declares local a function previously declared externally visible from this module.	Remove the inappropriate declaration.
field selection from incomplete type not allowed	Code refers to a field selector (member) of an incomplete struct type.	Complete the struct type or remove the field selector reference.
identifier-list parameters may only be used in a function definition	Using K&R declaration syntax inappropriately.	Use prototypes when declaring a function or do not use a parameter identifier list for K&R-syntax function declarations.
improperly terminated macro invocation	One or more missing close parentheses.	Use correct number of close parentheses in macro calling statements.
incomplete type not allowed	Code contains array type of unknown size, structure type of unknown content, or union type of unknown content in an inappropriate context.	Make sure each type is complete before code refers to it.
invalid use of non-lvalue array	Code refers to a non-lvalue array, but context requires an lvalue array.	Change code, to refer to an lvalue array.

Table A-1. MECC Error Messages (cont.)

Message	Probable Cause	Corrective Action
out of memory	Compiling the program at the requested optimization level requires additional memory.	Use the **`-Amemlimit`** control variable to increase available memory, or lower the optimization level.
pointer initialized to address of position independent object.	Code initializes a pointer to be the address of a position-independent object.	Avoid initializing a pointer to the address of a position-independent object. Make such an assignment in subsequent code, not in declarations.
this entity may not be initialized	Code specifies an initializer for an object that cannot be initialized, such as a parameter or an object with an incomplete type.	Do not attempt to initialize the object.
translation unit must contain at least one declaration	Source file lacks any declarations.	Include at least one declaration in the source file.

APPENDIX B
LANGUAGE DIALECTS

This appendix describes the three C dialects that the compiler accepts: ANSI, Kernighan and Ritchie, and relaxed. The extensions, differences, or constructs of each dialect fall into these standard categories:

1. Preprocessor
2. Variable/type declarations
3. Syntax
4. Pointer semantics
5. Expressions
6. Extensions
7. Extended operations

B.1 ANSI C

The compiler accepts the ANSI C language defined by X3.159-1989 plus these extensions:

1: Preprocessor:

- The compiler accepts comment text at the end of preprocessing directives.

2: Variable/type declarations:

- The compiler allows bit-fields with base types that are **`enums`** or integral types other than **`int`** and **`unsigned int`**.
- The compiler accepts a multi-member **`struct`** in which the last member has an incomplete array type.
- The compiler lets a file-scope array have an incomplete struct or union type as its element type. However, the struct or union type must be completed before the array is subscripted (if ever); the type must be completed by the end of compilation if the array is not external.
- The compiler accepts static function declarations in function and block scopes, moving these declarations to the file scope.
- The compiler accepts incomplete **`enum`** tags, defining and resolving tag names later.

- The compiler lets you redeclare **`typedef`** names in the same scope, but issues a warning message.

3: Syntax:

- The compiler accepts an empty declaration (a semicolon with nothing preceding it), but issues a warning message.
- The compiler lets single-value expressions, not enclosed in braces, initialize an entire static array, **`struct`**, or **`union`**.
- The compiler accepts **`long float`** as a synonym for **`double`**.
- The compiler accepts pointers to incomplete arrays for pointer addition, subtraction, and subscripting, as in this example:

```
int (*p)[];
...
q = p[0];
```

  If the added or subtracted value is not a constant zero, the compiler issues a warning message. (The compiler multiplies the value by zero, as the pointer is to a type of zero size, so the operation does not affect the result.)
- The compiler allows an extra comma at the end of the `enum` list, but issues a warning message.
- The compiler lets you omit the final semicolon before the close brace (`}`) of a `struct` or `union` specifier, but issues a warning message.
- You may follow label definitions immediately with a close brace (`}`) instead of a statement, but the compiler issues a warning message.

4: Pointer semantics:

- The compiler lets an initializer pointer constant value be cast to an integral type, if the integral type is sufficiently large.
- The compiler lets an integer constant, of an integral constant expression, be cast to a pointer type, then be recast as an integer.
- You may assign pointers to interchangeable but different types, such as `unsigned char *` and `char *`. Similarly, the compiler allows differences between interchangeable but different pointer types. Eligible pointers include those to same-sized integral types (typically `int *` and `long *`). The compiler issues a warning message, unless you assign a string constant to a pointer of any kind of character.

- You may assign pointer types if the destination type has added second- or lower-level qualifiers (such as `int **` to `const int **`), but the compiler issues a warning message. Similarly, you may do comparison or pointer-difference operations on such pairs of pointers, but the compiler issues a warning message.
- In pointer operations, the compiler implicitly converts a pointer to `void` to another type whenever such conversion is required. Similarly, the compiler implicitly converts a null pointer constant to a null pointer of the correct type whenever such conversion is required.
- For pointers to functions of different types, the compiler allows assignment, equality comparisons (`==`), or inequality comparisons (`!=`) without an explicit type cast, but issues a warning message.
- The compiler lets a pointers to `void` be implicitly converted to or from a pointer to a function type, but issues a warning message.

6: extensions:

- The compiler accepts asm statements and declarations, provided that your command line specifies **`-Ac=ansi`**, then **`-Aasm=1`**.

7: Extended operations:

- The compiler evaluates the expression of **`__INTADDR__ (expression)`** as a constant expression, then converts it to an integer constant.
- **`ALIGNOF__`** is similar to **`sizeof`**, but returns the type alignment value. (If there is no alignment requirement, **`__ALIGNOF__`** returns the value 1.) A type or expression in parentheses may follow **`__ALIGNOF__`**:

  Type: **`__ALIGNOF__(int)`**

  Expression: **`__ALIGNOF__(a*i+4)`**

  Note that the compiler determines the alignment of the expression (a*i+4), but does not evaluate the expression.

Also note that the compiler considers the value of a character or string escape to be the character itself, provided that the character following the backslash (`\`) has no special meaning. (X3.159-1989 does not clearly define this situation.)

B.2 K&R C

In K&R mode, the compiler accepts the C language Kernighan and Ritchie defined in *The C Programming Language* (Prentice-Hall, 1978). Note that the K&R mode does not disable ANSI C features that do not conflict with K&R C.

The list below highlights specific differences between K&R C and ANSI C:

1: Preprocessor:

- The compiler does not recognize trigraphs.
- The compiler begins searching for include files in the directory holding the file that contains **`#include`**, rather than in the directory containing the primary source file.
- The compiler deletes comments in preprocessing output instead of replacing them with a space.
- The compiler does not recognize the **`#`** or **`##`** operator in macro definitions.
- The compiler does not recognize the escape character **`\a`** (alert) in character or string constants.
- The compiler does not macro expand arguments to macros before inserting them into the macro expansion. When the compiler rescans the macro expansion it expands macro invocations in the argument text. This allows for macro recursion.
- The compiler does not maintain end-of-marker tokens: tokens that abut after macro substitution are parsed as single tokens.
- The compiler ignores multiple **`#else`** directives in an **`#if`** block, but generates a warning diagnostic message.
- If a macro parameter name is inside a character or string constant, the compiler recognizes and expands that macro parameter.
- The compiler generates warning messages instead of error messages for macro invocations with too many arguments. The compiler ignores the extra arguments.
- The compiler lets you leave undefined the standard preprocessor symbol **`__STDC__`**.
- The compiler does not generate extra spaces in textual preprocessing output; this permits pasting of adjacent tokens.

2: Variable/type declarations:

- Any declaration of external functions and variables is visible to the rest of the file.
- For function parameter lists beginning with a **`typedef`** identifier, the compiler does not consider the parameter list to be prototyped unless something (except a comma or close parenthesis) follows the **`typedef`** identifier, as in this code fragment:

```
typedef int t;
int f(t) { } /*old style list*/
int g(t x) { } /*prototyped list*/
```

  In ANSI C mode, the compiler would generate an error on the first example, as ANSI C considers any parameter list starting with a **`typedef`** identifier to be prototyped.
- If a file-scope array has an unspecified storage class and remains incomplete at the end of compilation, the compiler considers the array's storage class to be **`extern`**.
- In the declaration of a member of a **`struct`** or **`union`**, the compiler lets you omit a declarator list to specify an unnamed (non-bit field) field for padding, as in:

```
struct s {int a; int; int b;} v;
```

- If you declare a function **`static`** but never use the function, the compiler considers the function to have **`extern`** storage class.
- The compiler does not generate any warning message if storage specifiers appear in a list of specifiers in any position other than first.
- The compiler gives **`enum`** keywords the default type **`int`**. The compiler uses smaller integral types if the **`-Aminsizenums`** flag is set (that is, given the value 1).
- The compiler treats **`short`**, **`long`**, and **`unsigned`** as *adjectives* in type specifiers; you must use an adjective to modify a **`typedef`** type.
- The compiler promotes **`float`** functions and parameters to **`double`** functions and parameters.
- The compiler lets you omit declaration specifiers:

```
i;
```

  declares **`i`** to be an **`int`** variable. The compiler does issue a warning message for such a default declaration. (ANSI C lets you omit specifiers only for function declarations.)
- The compiler lets identifiers in a function and parameters of the function have the same name, but issues a warning message.
- The compiler promotes **`unsigned char`** and **`unsigned short`** to unsigned **`int`**.

3: Syntax

- The compiler disables the ANSI C keywords **signed**, **const**, and **volatile** to avoid conflicts. (The compiler does not disable keywords **enum** and **void**.)
- The compiler ignores declarations of the form:

```
typedef some-type void;
```

- The compiler allows field selections of the form **p->field**, even if **p** does not point to a **struct** or **union** that contains **field**. If **x** is an **lvalue**, the compiler allows **x.field** even if **x** is not a **struct** or **union** that contains **field**. Each definition of **field** as a field must have the same offset in its respective **struct** or **union**.
- The compiler issues a warning message if you apply an ampersand (**&**) to an array. The ampersand operation is *type address of array element*, (not *address of array*.)

4: Pointer semantics:

- The compiler allows assignment between pointers and integers, and between incompatible pointer types, without explicit casts (but issues a warning message). You may assign a pointer to an integer, unless the integer is smaller than the pointer.
- The compiler does not let you share string literals. Identical string literals produce multiple copies of the string to be allocated.

5: Expressions:

- The compiler performs the usual arithmetic conversions for the shift operators (**<<** and **>>**): it converts the right operand to **int**; the result type is the left-operand type. This means that a **long** shift count forces the shift to be done as **long**. (In ANSI C mode, the compiler does integral promotions on both operands; the result type is the left-operand type.)
- The compiler maintains an **lvalue** cast to a type of the same size as that particular **lvalue**, unless the **lvalue** requires a floating-point conversion.
- The compiler lets you apply **sizeof** to bit-fields; the size is the size of the underlying type.
- The compiler interprets all **float** operations as **double**.
- The compiler considers plain **char** to be the same as **signed char** or **unsigned char**, according to the default and command-line control options. (In ANSI C, **char** is a distinct type.)
- The compiler accepts **0x** as a hexadecimal zero and generates a warning message.
- If the compiler finds the digits **8** or **9** in any octal constants, it treats the constant as decimal.
- The compiler accepts **1E+** as floating-point constant with an exponent of zero (and issues a warning message).

- The compiler accepts an integer constant larger than can be stored in an **`unsigned long`**, but truncates the value to an appropriate number of low-order bits. The compiler issues a warning message.
- The compiler sets the type of a large integer constant according to K&R rules: it does not assign **`unsigned`** in some cases where ANSI C would. The compiler types integer constants with apparent values larger than **`LONG_MAX`** as **`long`**. To suppress related warning messages, the compiler considers such constants *non-arithmetic*.

B.3 RELAXED C

In relaxed C mode (the default), the compiler accepts a less-strict variant of ANSI C. The compiler accepts minor violations of the ANSI C standard (but issues warning messages.) Additionally, it accepts these IBM AIX C constructs:

1: Preprocessor:

- The compiler generates warning messages instead of error messages for macro invocations with too many arguments. The compiler ignores the extra arguments.

2: Variable/type declarations:

- The compiler lets a function declared to return **`void`** be redeclared to return **`int`**, and vice versa. For example, the compiler accepts this function redeclaration:

```
extern void some_other_routine(int arg1, int arg2);
extern int some_other_routine(int arg1, int arg2);
```

  (The compiler issues a warning message about the second declaration).
- The compiler accepts unnamed **`struct`** and **`union`** field members. You can use unnamed members for padding within a **`struct`** or **`union`**.
- The compiler ignores the creation a **`typedef`** named **`void`**. This supports pre-ANSI C code.

```
typedef int void;
```

- The compiler accepts function declarations with no storage class or type specifier, such as:

```
a();  /* ANSI C error; Relaxed warning */
void main (void)
{
   a(12);
}
```

3: Syntax

- The compiler lets you use the structure member operator (**.**) or structure pointer operator (**->**) with a non-**struct** variable, provided that the **struct**-member name used is unique. An example is:

```
void main (int bbb)
{
  struct fred {
                        int f1;
                        char f2;
                        int f3;
  }                   foo;

int a, b;
  a = 12;
  bbb = 999;
  bbb.f2 = 10; /* bbb is not a struct */
}
```

(If another **struct** also had a member named **f2**, the compiler would not have accepted this code.)

4: Pointer semantics:

- The compiler lets you mix pointer and **int** arithmetic, but generates a warning message.
- The compiler allows assignment between incompatible pointer types, but generates a warning message.

5: Expressions:

- The compiler lets you use the C built-in **`sizeof()`** on bit-fields: **`sizeof(some_bit_field)`** returns the bit-field type size. For example, in this code excerpt:

```
struct abc {
   int field1;
   int field2:24;
   char field3:8
}  some_struct;
```

  the value of **`sizeof(some_struct.field2)`** is 4; the value of **`sizeof(some_struct.field3)`** is 1.

- The compiler allows a **`const`** qualifier on the right-hand side of an expression, but generates a warning message.
- The compiler relaxes type-qualifier checking when determining compatibility of two types.
- The compiler promotes **`unsigned char`** and **`unsigned short`** to **`unsigned int`**.

APPENDIX C
C RUN-TIME LIBRARIES

The software for your Motorola Embedded PowerPC™ Compiler Package includes the Motorola embedded PowerPC C run-time libraries:

- **`libppc.a`** — ANSI C routines for code or data that is not position independent,
- **`libppcp.a`** — ANSI C routines for code or data that *is* position independent,
- **`libsys.a`** — support routines for code or data that is not position independent, and
- **`libsysp.a`** — support routines for code or data that *is* position independent.

C.1 ANSI C ROUTINES

The ANSI C routines in libraries **`libppc.a`** and **`libppcp.a`** are based on P. J. Plauger's C library source, version 2.2.1. These libraries contain identical routines, except that routines of **`libppcp.a`** are for position-independent code or data.

NOTE

Routines of **`libppcp.a`** support the position independent code you create by compiling with the **`pic`** control variable. The routines support the position independent data you create by compiling with the **`pid`** control variable.

Library file **`libppc.a`** is an archive of object files. The compiler-package software also includes the ANSI C routines as a trees of source files and header files that generate the object files. A read-me file (**`README.libppc`**) and library version file (**`version_libppc.src`**) accompany the source files and header files. A copy of file **`version_libppc.src`** (**`version_libppc.txt`**) appears inside the archive. (In the corresponding files for **`libppcp.a`**, the string **`ppcp`** replaces **`ppc`** in the file names.)

Table C-1 lists the ANSI C library functions and macros that remain available if your **`mecc`** command line includes the **`-DEMB_PPC`** option. You may need to edit these functions and macros before including them in your application.

Table C-2 lists the ANSI C library functions and macros *not* available if your **`mecc`** command line includes the **`-DEMB_PPC`** option. These are functions and macros that depend on host operating-system services: time and date, file system and input/output, dynamic (heap) memory management, signals, or process management.

In general, if you use the **-DEMB_PPC** option to compile and link source files to form executables, the source files should include **libppc.a**'s (or **libppcp.a**'s) header files. Furthermore, you also should use the **-DEMB_PPC** option to compile the header files.

Table C-1. Functions/Macros Available Via the -DEMB_PPC Option

abs	**fmod**	**longjmp**	**sinh**	**strspn**
acos	**frexp**	**log**	**sprintf**(2)	**strstr**
asin	**isalnum**	**log10**	**sqrt**	**strtod**
atan	**isalpha**	**memchr**	**srand**	**strtok**
atan2	**iscntrl**	**memcmp**	**strcat**	**strtol**
atof	**isdigit**	**memcpy**	**strchr**	**strtoul**
atoi	**isgraph**	**memmove**	**strcmp**	**tan**
atol	**islower**	**memset**	**strcpy**	**tanh**
bsearch	**isprint**	**modf**	**strcspn**	**tolower**
ceil	**ispunct**	**offsetof**	**strerror**	**toupper**
cos	**isspace**	**pow**	**strlen**	**va_arg**
cosh	**isupper**	**printf**(1)(2)	**strncat**	**va_end**
div	**isxdigit**	**qsort**	**strncmp**	**va_start**
exp	**labs**	**rand**	**strncpy**	**vprintf**(1)(2)
fabs	**ldexp**	**setjmp**	**strpbrk**	**vsprintf**(2)
floor	**ldiv**	**sin**	**strrchr**	

(1) These functions require linkage with the user-defined function **_User_defined_printf_aux()**. The prototype of this function appears in function **stdio.h** if your **mecc** command line includes the **-DEMB_PPC** option. Function **_User_defined_printf_aux()** should deliver the number of characters its third argument specifies; it should store those characters in the buffer its second argument points to. **_User_defined_printf_aux()** should return its first-argument value upon success, or the value zero upon failure.

(2) These functions do not support multibyte or wide characters in this implementation. Code for such support is in the source (when EMB_PPC is not defined), but requires dynamic memory allocation and environment operating system services.

Table C-2. Functions/Macros Not Available Via the -DEMB_PPC Option

abort	**fgetc**	**fwrite**	**putc**	**strcoll**
asctime	**fgetpos**	**getc**	**putchar**	**strftime**
assert	**fgets**	**getchar**	**puts**	**strxfrm**
atexit	**fopen**	**getenv**	**raise**	**system**
calloc	**fprintf**	**gets**	**realloc**	**time**
clearerr	**fputc**	**gmtime**	**remove**	**tmpfile**
clock	**fputs**	**localeconv**	**rename**	**tmpname**
ctime	**fread**	**localtime**	**rewind**	**ungetc**
difftime	**free**	**malloc**	**scanf**	**vprintf**
exit	**freopen**	**mblen**	**setbuf**	**wcstombs**
fclose	**fscanf**	**mbstowcs**	**setlocale**	**wctomb**
feof	**fseek**	**mbtowc**	**setvbuf**	
ferror	**fsetpos**	**mktime**	**signal**	
fflush	**ftell**	**perror**	**sscanf**	

C.2 SUPPORT ROUTINES

The support routines in libraries **libsys.a** and **libsysp.a** contain identical routines, except that routines of **libsysp.a** are for position-independent code or data.

NOTE

Routines of **libsysp.a** support the position independent code you create by compiling with the **pic** control variable. The routines support the position independent data you create by compiling with the **pid** control variable.

Library file **libsys.a** is an archive of object files. The compiler-package software also includes the service routines as a trees of source files and header files that generate the object files. A library version file (**version.txt**) appears inside the archive. (A corresponding library version file appears inside file **libsysp.a**.)

Table C-3 lists the service routines. All but three of these routines have exceptions; the description text notes the three routines that do not. Note that **a** and **b**, in the description text denote actual values, not addresses.

Table C-3. System Library Support Routines

Function Prototypes	Description
long double _q_add(const long double *a, const long double *b)	Returns **a** + **b**.
int _q_cmp(const long double *a, const long double *b)	Performs an unordered comparison of the values of **a** and **b**. Returns int 0 if **a** = **b**, int 1 if **a** < **b**, int 2 if **a** > **b**, int 3 if **a** is unordered with respect to **b**.
int _q_cmpe(const long double *a, const long double *b)	Performs an ordered comparison of the values of **a** and **b**. Returns int 0 if **a** = **b**, int 1 if **a** < **b**, int 2 if **a** > **b**, int 3 if **a** is unordered with respect to **b**.
long double _q_div(const long double *a, const long double *b)	Returns **a** / **b**.
long double _q_dtoq(double a)	Converts the value of **a** to a long double value. Returns that value.
int _q_feq(const long double *a, const long double *b)	Performs an unordered comparison of the values of **a** and **b**. Returns a nonzero value if **a** = **b**; returns a zero if **a** ≠ **b**.
int _q_fge(const long double *a, const long double *b)	Performs an ordered comparison of the values of **a** and **b**. Returns a nonzero value if **a** >= **b**; returns a zero if **a** < **b**.
int _q_fgt(const long double *a, const long double *b)	Performs an ordered comparison of the values of **a** and **b**. Returns a nonzero value if **a** > **b**; returns a zero if **a** <= **b**.
int _q_fle(const long double *a, const long double *b)	Performs an ordered comparison of the values of **a** and **b**. Returns a nonzero value if **a** <= **b**; returns a zero if **a** > **b**.
int _q_flt(const long double *a, const long double *b)	Performs an ordered comparison of the values of **a** and **b**. Returns a nonzero value if **a** < **b**; returns a zero if **a** >= **b**.

Table C-3. System Library Support Routines (continued)

Function Prototypes	Description
int _q_fne(const long double *a, const long double *b)	Performs an unordered comparison of the values of **a** and **b**. Returns a nonzero value if **a** = **b** or if they are unordered. Otherwise, returns a zero.
long double _q_itoq(int a)	Converts the int value of **a** to a long double value. Returns that value. Does not have any exceptions.
long double _q_mul(const long double *a, const long double *b)	Returns **a** * **b**.
long double _q_neg(const long double *a)	Returns -**a** . Does not have any exceptions.
double _q_qtod(const long double *a)	Converts the value of **a** to a long double value. Returns that value.
int _q_qtoi(const long double *a)	Converts the value of **a** to a signed int value by truncating any fractional part. Returns that value.
float _q_qtos(const long double *a)	Converts the value of **a** to a float value. Returns that value.
unsigned int _q_qtou(const long double *a)	Converts the value of **a** to an unsigned int value by truncating any fractional part. Returns that value.
long double _q_sqrt(const long double *a)	Returns the square root of **a**.
long double __stoq(float a)	Converts the value of **a** to a long double value. Returns that value.
long double _q_sub(const long double *a, const long double *b)	Returns **a** - **b**.
long double _q_utoq(unsigned int a)	Converts the value of **a** to a long double value. Returns that value. Does not have any exceptions.
unsigned int __dtou(double a)	Converts the value of **a** to an unsigned int value by truncating any fractional part. Returns that value.

INDEX