CodeWarrior Development Studio for Microcontrollers V10.x Digital Signal Controller Build Tools Reference Manual

Document Number: CWMCUDSCCMPREF
Rev 10.6, 02/2014

Contents

Section number **Title** **Page**

Chapter 1
Introduction

Chapter 2
Using Build Tools with the CodeWarrior IDE

Chapter 3
Using Build Tools on the Command Line

Chapter 4
C for DSP56800E

Chapter 5
C Compiler

Chapter 6
C++ Compiler

Chapter 7
ELF Linker

Chapter 8
Inline Assembly Language and Intrinsics

Chapter 9
Pragmas

Chapter 10
Predefined Symbols

Chapter 11
Optimization

Chapter 12
Tool Performance

Chapter 13
Libraries and Runtime Code

Chapter 1
Introduction

This reference manual describes how to use the CodeWarrior compiler and linker tools to build software.

CodeWarrior build tools are programs that translate source code into object code then organize that object code to create a program that is ready to execute.

CodeWarrior build tools often run on a different platform than the programs they generate. The host platform is the machine on which CodeWarrior build tools run. The target platform is the machine on which the software generated by the build tools runs.

This section introduces how CodeWarrior build tools are organized:

1.1 Compiler Architecture

From your perspective, a CodeWarrior compiler is a single program. Internally, however, a CodeWarrior compiler has two parts:

- The front-end, shared by all CodeWarrior compilers, translates human-readable source code into a platform-independent intermediate representation of the program being compiled
- The back-end, customized to generate software for a target platform, converts the intermediate representation into object code containing data and native instructions for the target processor

A CodeWarrior compiler coordinates its front-end and back-end to translate source code into object code in several steps:

- Configure settings requested from the compiler to the CodeWarrior IDE or passed to the linker from the command-line
- Translate human-readable source code into an intermediate representation

- Optionally output symbolic debugging information
- Optimize the intermediate representation
- Convert the intermediate representation to native object code
- Optimize the native object code
- Output the native, optimized object code

1.2 Linker Architecture

A linker combines and arranges the object code in libraries and object code generated by compilers and assemblers into a single file or image, ready to execute on the target platform. The CodeWarrior linker builds an executable image in several steps:

- Configure settings requested from the linker to the CodeWarrior IDE or passed to the linker from the command-line
- Read settings from a linker control file
- Read object code
- Search for and ignore unused objects ("deadstripping")
- Build and output the executable file
- Optionally output a map file

Chapter 2
Using Build Tools with the CodeWarrior IDE

The CodeWarrior Integrated Development Environment (IDE) uses settings in a project properties: C/C++ Build >> Settings >> Tools Settings. Each individual tool has its own settings group as such: DSC Linker, DSC Compiler, DSC Assembler, DSC preprocessor, DSC disassembler.

This chapter describes how to use CodeWarrior compilers and linkers with the CodeWarrior IDE:

2.1 IDE Options and Pragmas

The build tools determine their settings by IDE settings and directives in source code.

The CodeWarrior compiler follows these steps to determine the settings to apply to each file that the compiler translates under the IDE:

- before translating the source code file, the compiler gets option settings from the IDE's settings panels in the current build target
- the compiler updates the settings for pragmas that correspond to panel settings
- the compiler translates the source code in the **Prefix Text** field of the build target's **C/C++ Preprocessor** panel

  The compiler applies pragma directives and updates their settings as pragmas directives are encountered in this source code.

- the compiler translates the source code in the Prefix Text field of the build target's DSC Compiler >> Input panel

  The compiler applies pragma directives and updates their settings as pragmas are encountered.

2.2 Build Properties for DSC

The following image shows the **Properties for** <*project*> dialog box displaying the corresponding build properties for DSC CPU project.

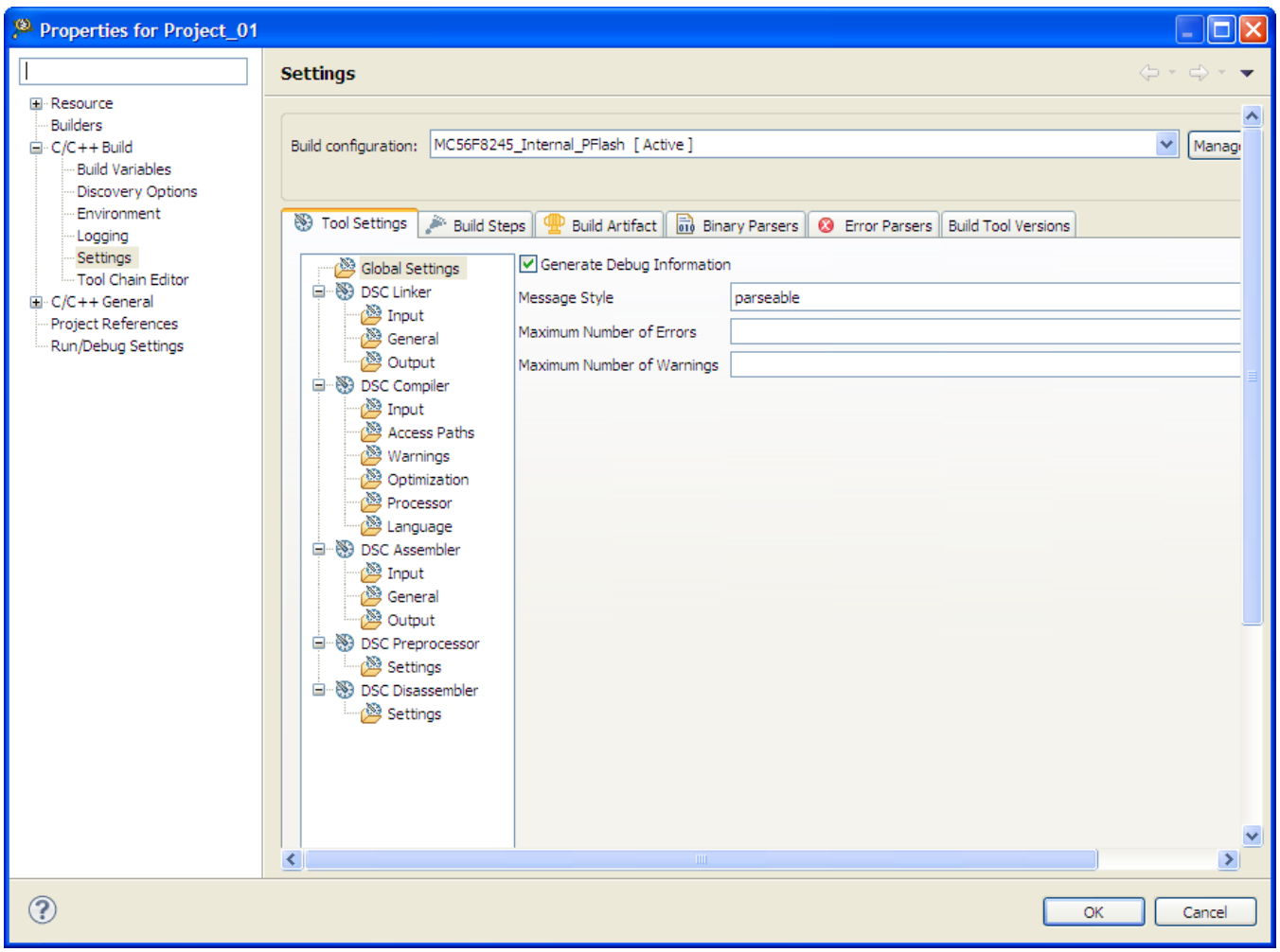

Figure 2-1. Build Properties - DSC

The following table lists the build properties specific to developing software for DSC.

The properties that you specify in the **Tool Settings** panels apply to the selected build tool on the **Tool Settings** page of the **Properties for** <*project*> window.

Table 2-1. Build Properties for DSC

Build Tool	Build Properties Panels
Global Settings	Global Settings
DSC Compiler	DSC Compiler > Input

Table continues on the next page...

Table 2-1. Build Properties for DSC (continued)

Build Tool	Build Properties Panels
	DSC Compiler > Access Paths
	DSC Compiler > Warnings
	DSC Compiler > Optimization
	DSC Compiler > Processor
	DSC Compiler > Language
DSC Assembler	DSC Assembler > Input
	DSC Assembler > General
	DSC Assembler > Output
DSC Linker	DSC Linker > Input
	DSC Linker > General
	DSC Linker > Output
DSC Preprocessor	DSC Preprocessor > Settings
DSC Disassembler	DSC Disassembler > Settings

2.2.1 Global Settings

Use this panel to specify the global settings the DSC architecture uses. The build tools (compiler, linker, and assembler) then use the properties set in this panel to generate CPU-specific code.

The following table lists and describes the global settings options for DSC.

Table 2-2. Tool Settings - Global Settings

Option	Description
Generate Debug Information	Check to generate symbolic information for debugging the build target.
Message Style	List options to select message style. • **GCC(default)** - Uses the message style of the Gnu Compiler Collection tools • **MPW** - Uses the Macintosh Programmer's Workshop (MPW®) message style • **Standard** - Uses the standard message style • **IDE** - Uses context-free machine parseable message style • **Enterprise-IDE -** Uses CodeWarrior's Integrated Development Environment (IDE) message style. • **Parseable** - Uses parseable message style.
Maximum Number of Errors	Specify the number of errors allowed until the application stops processing.
Maximum Number of Warnings	Specify the maximum number of warnings.

2.2.2 DSC Linker

Use this panel to specify the DSC linker behavior. You can specify the command, options, and expert settings for the build tool linker. Additionally, the Linker tree control includes the input, general, and output settings.

The following table lists and describes the linker options for DSC.

Table 2-3. Tools Settings > DSC Linker Options

Option	Description
Command	Shows the location of the linker executable file. You can specify additional command line options for the linker; type in custom flags that are not otherwise available in the UI.
All options	Shows the actual command line the linker will be called with.
Expert settings	Shows the expert settings command line parameters; default is `${COMMAND} ${FLAGS} ${OUTPUT_FLAG} ${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}`.
Command line pattern	

2.2.2.1 DSC Linker > Input

Use this panel to specify files the DSC linker should use. You can specify multiple additional libraries and library search paths. Also, you can change the order in which the IDE uses or searches the libraries.

The following table lists and describes the linker input options for DSC.

Table 2-4. Tools Settings > DSC Linker > Input

Option	Description
No Standard Library	Check if you do not want to include the standard library.
Linker Command File	Consists of three kinds of segments, which must be in this order: • A memory segment, which begins with the MEMORY{} directive. • Optional closure segments, which begin with the FORCE_ACTIVE{}, KEEP_SECTION{}, or REF_INCLUDE{} directives. • A sections segment, which begins with the SECTIONS{} directive.
Entry Point	Specifies the program starting point: the first function the debugger uses upon program start; default: __start. This default function is in file Finit_MC56F824x_5x_ISR_HW_RESET. It sets up the DSC environment before code execution. Its final task is calling main().

Table continues on the next page...

Table 2-4. Tools Settings > DSC Linker > Input (continued)

Option	Description
Library Search Paths (-L)	Lets you add/update the search pathname of libraries or other resources related to the project. Click the **Add** button and type the pathname into the **Directory** text box. Alternatively, click **Workspace** or **File system** , then use the subsequent dialog box to browse to the correct location.
Library Recursive Search Paths (-lr)	Lets you add/update the recursive search pathname of libraries or other resources related to the project. Click the **Add** button and type the pathname into the **Directory** text box. Alternatively, click **Workspace** or **File system** , then use the subsequent dialog box to browse to the correct location.
Additional Libraries	Specify multiple additional libraries and library search paths. Also, you can change the order in which the IDE uses or searches the libraries.
Force Active Symbols	Directs the linker to include symbols in the link, even if those symbols are not referenced. Makes symbols immune to deadstripping. Separates multiple symbols with single spaces.

2.2.2.2 DSC Linker > General

Use this panel to specify the general linker behavior.

The following table lists and describes the linker options for DSC.

Table 2-5. Tools Settings > DSC Linker > General

Option	Description
Dead-Strip Unused Code	Determines whether to pool constants from all functions in a file.
Suppress Link Warnings	Prevents the IDE from displaying linker warning messages.
Large Data Memory Model	Check to extend the DSP56800E addressing range by providing 24-bit address capability to instructions. Clear if you do not want to extend the address range.
Generates elf file for 56800EX core	Check to generate elf file for 56800EXcore that makes a program file out of the object files of your project. The linker also allows you to manipulate code in different ways. You can define variables during linking, control the link order to the granularity of a single function, change the alignment, and even compress code and data segments so that they occupy less space in the output file. All of these functions are accessed through commands in the linker command file (LCF). The linker command file has its own language complete with keywords, directives, and expressions, that are used to create the specifications for your output code. The syntax and structure of the linker command file is similar to that of a programming language.
Other Flags	Specify additional command line options for the linker; type in custom flags that are not otherwise available in the UI.

2.2.2.3 DSC Linker > Output

Use this panel to specify the output settings for the DSC linker output.

The following table lists and describes the linker options for DSC.

Table 2-6. Tools Settings > DSC Linker > Output

Option	Description
Output Type	Select application as Application (default), Library, or Partial Linking.
Generate Link Map	Check to generate link map.
List Unused Symbols in Map	Check to list unused symbols; appears grayed out if the Generate Link Map checkbox is not checked.
Show Transitive Closure in Map	Check show transitive closure; appears grayed out if the Generate Link Map checkbox is not checked.
Annotate Byte Symbols in Map	Check if you want the linker to include B annotation for byte data types (e.g., char) in the Linker Command File. By default, the Linker does not include the B annotation in the Linker Command File. Everything without the B annotation is a word address.
Generate ELF Symbol Table	Check to generated the ELF symbol table.
Generate S-Record File	Check to generate a S-record file.
Sort by Address	Check to sort by address.
Generate Byte Addresses	Check to generate byte address.
Max S-Record Length	Specify the maximum length for S-record; appears grayed out if the Generate S-Record File checkbox is not checked. The default value is 252.
S_Record EOL Character	Specify the end-of-line character; appears grayed out if the Generate S-Record File checkbox is not checked. The default value is DOS (\\r\\n).

2.2.3 DSC Compiler

Use this panel to specify the command, options, and expert settings for the build tool compiler. Additionally, the DSC Compiler tree control includes the general and the file search path settings.

The following table lists and describes the linker options for DSC.

Table 2-7. Tools Settings > DSC Compiler

Option	Description
Command	Shows the location of the compiler executable file. You can specify additional command line options for the compiler; type in custom flags that are not otherwise available in the UI.
All options	Shows the actual command line the compiler will be called with.
Expert settings	Shows the expert settings command line parameters; default is `${COMMAND} -c ${FLAGS} ${OUTPUT_FLAG} ${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}`.
Command line pattern	

2.2.3.1 DSC Compiler > Input

Use this panel to specify additional files the DSC Compiler should use. You can specify multiple additional libraries and library search paths. Also, you can change the order in which the IDE uses or searches the libraries.

The following table lists and describes the compiler inputs options for DSC.

Table 2-8. Tools Settings > DSC Compiler > Input

Option	Description
Prefix File	Specifies a file to be included at the beginning of every assembly file of the project. Lets you include common definitions without using an include directive in every file.
Source File Encoding	Allows you to specify the default encoding of source files. Multibyte and Unicode source text is supported.
Allow Macro Redefinition	Enables to redefine the macros with the #define directive without first undefining them with the #undef directive.
Defined Macros	Lists the defined command-line macros.
Undefined Macros	Lists the undefined command-line macros.

2.2.3.2 DSC Compiler > Access Paths

Use this panel to specify the access paths. Access paths are directory paths the CodeWarrior tools use to search for libraries, runtime support files, and other object files.

The following table lists and describes the compiler access paths for DSC.

Table 2-9. Tools Settings > DSC Compiler > Access Paths

Option	Description
Search User Paths (#include "...")	Lets you add/update the user paths that the CodeWarrior IDE searches to find files in your project. You can add several kinds of paths including absolute and project-relative.
Search User Paths Recursively	Lets you add/update the recursive user paths that the CodeWarrior IDE searches to find files in your project. You can add several kinds of paths including absolute and project-relative.
Search System paths (#include <...>)	Lets you add/update the system paths that the CodeWarrior IDE searches to find files in your project. You can add several kinds of paths including absolute and project-relative.
Search System Paths Recursively	Lets you add/update the recursive system paths that the CodeWarrior IDE searches to find files in your project. You can add several kinds of paths including absolute and project-relative.

2.2.3.3 DSC Compiler > Warnings

Use this panel to control how the DSC compiler formats the listing file, error and warning messages.

The following table lists and describes the compiler warnings options for DSC.

Table 2-10. Tool Settings - DSC Compiler > Warnings

Option	Description
Treat All Warnings As Errors	Check to treat all warnings as errors. The compiler will stop if it generates a warning message.
Enable Warnings	Select the level of warnings you want reported from the compiler. Custom lets you to select individual warnings. Other settings select a pre-defined set of warnings.
Illegal #Pragmas (most)	Check to notify the presence of illegal pragmas.
Possible Unwanted Effects (most)	Check to notify most of the possible errors.
Extended Error Checks (most)	Check if you want to do an extended error checking.
Hidden virtual functions (most)	Check to generate a warning message if you declare a non-virtual member function that prevents a virtual function, that was defined in a superclass, from being called and is equivalent to pragma `warn_hidevirtual` and the command-line option `-warnings hidevirtual`.
Implicit Arithmentic Conversions (all)	Check to warn of implict arithmetic conversions.
Implicit Signed/Unsigned Conversion (all)	Check to enable warning of implict conversions between signed and unsigned variables.
Implicit Float to Integer Conversions (all)	Check to warn of implict conversions of a floating-point variable to integer type.
Implicit Integer to Float Conversions (all)	Check to warn of implict conversion of an integer variable to floating-point type.

Table continues on the next page...

Table 2-10. Tool Settings - DSC Compiler > Warnings (continued)

Option	Description
Pointer/Integer Conversions (most)	Check to enable warnings of conversions between pointer and integers.
Unused Arguments (most)	Check to warn of unused arguments in a function.
Unused Variables (most)	Check to warn of unused variables in the code.
Unused Result From Non-Void-Returning Function (full)	Check to warn of unused result from non-void-returning functions.
Missing `return' value in Non-Void-Returning Function (most)	Check to warn of when a function lacks a return statement.
Expression Has No Side Effect (most)	Check to issue a warning message if a source statement does not change the program's state. This is equivalent to the pragma `warn_no_side_effect`, and the command-line option `-warnings unusedexpr`.
Extra Commas (most)	Check to issue a warning message if a list in an enumeration terminates with a comma. The compiler ignores terminating commas in enumerations when compiling source code that conforms to the ISO/IEC 9899-1999 ("C99") standard and is equivalent to pragma `warn_extracomma` and the command-line `option -warnings extracomma`.
Empty Declarations (most)	Check to warn of empty declarations.
Inconsistent `class' / `struct' Usage (most)	Check to warn of inconsistent usage of class or struct.
Incorrect Capitalization in #include "..." (most)	Check to issue a warning message if the name of the file specified in a #include "file" directive uses different letter case from a file on disk and is equivalent to pragma `warn_filenamecaps` and the command-line option `-warnings filecaps`.
Incorrect Capitalization in System #Include <...> (most)	Check to issue a warning message if the name of the file specified in a #include <file> directive uses different letter case from a file on disk and is equivalent to pragma `warn_filenamecaps_system` and the command-line option `-warnings sysfilecaps`.
Pad Bytes Added (full)	Check to issue a warning message when the compiler adjusts the alignment of components in a data structure and is equivalent to pragma `warn_padding` and the command-line option `-warnings padding`.
Undefined Macro in #if/#elif (full)	Check to issues a warning message if an undefined macro appears in #if and #elif directives and is equivalent to pragma `warn_undefmacro` and the command-line option `-warnings undefmacro`.
Non-Inlined Functions (full)	Check to issue a warning message if a call to a function defined with the inline, __inline__, or __inline keywords could not be replaced with the function body and is equivalent to pragma `warn_notinlined` and the command-line option `-warnings notinlined`.
Token not formed by ## Operator (most)	Check to enable warnings for the illegal uses of the preprocessor's token concatenation operator (##). It is equivalent to the pragma `warn_illtokenpasting on`.

2.2.3.4 DSC Compiler > Optimization

Use this panel to control compiler optimizations. The compiler's optimizer can apply any of its optimizations in either global or non-global optimization mode. You can apply global optimization at the end of the development cycle, after compiling and optimizing all source files individually or in groups.

The following table lists and describes the compiler optimization options for DSC.

Table 2-11. Tool Settings - DSC Compiler > Optimization

Option	Description
Optimization Level	Specify the optimizations that you want the compiler to apply to the generated object code: • **Off (default)** - Disable optimizations. This setting is equivalent to specifying the `-opt level=0` command-line option. The compiler generates unoptimized, linear assembly-language code. • **1** - The compiler performs all target-independent (that is, non-parallelized) optimizations, such as function inlining. This setting is equivalent to specifying the `-opt level=1` command-line option. The compiler omits all target-specific optimizations and generates linear assembly-language code. • **2** - The compiler performs all optimizations (both target-independent and target-specific). This setting is equivalent to specifying the `-opt level=2` command-line option. The compiler outputs optimized, non-linear, parallelized assembly-language code. • **3** - The compiler performs all the level **2** optimizations, then the low-level optimizer performs global-algorithm register allocation. This setting is equivalent to specifying the `-opt level=3` command-line option. At this optimization level, the compiler generates code that is usually faster than the code generated from level **2** optimizations. • **4** - The compiler performs all the level **3** optimizations. This setting is equivalent to specifying the `-opt level=4` command-line option. At this level, the compiler adds repeated subexpression elimination and loop-invariant code motion.
Speed vs. Size	Use to specify an **Optimization Level** greater than **0** . • **Speed** - The compiler optimizes object code at the specified **Optimization Level** such that the resulting binary file has a faster execution speed, as opposed to a smaller executable code size. This setting is equivalent to specifying the `-opt speed` command-line option. • **Size** - The compiler optimizes object code at the specified **Optimization Level** such that the resulting binary file has a smaller executable code size, as opposed to a faster execution speed. This setting is equivalent to specifying the `-opt space` command-line option.

Table continues on the next page...

Table 2-11. Tool Settings - DSC Compiler > Optimization (continued)

Option	Description
Inter-Procedural Analysis	Control whether the compiler views single or multiple source files at compile time. • **Off-** Compiler compiles one file at a time. The functions are displayed in order as they appear in the source file. An object file is created for each source. • **File-** The compiler sees all the functions and data in a translation unit (source file) before code or data is generated. This allows inlining of functions that may not have been possible in -ipa off mode.
Inline Level	Enables inline expansion. If there is a #pragma INLINE before a function definition, all calls of this function are replaced by the code of this function, if possible. The options available are: • **Off** - No functions are inlined. • **Smart (default)** - Inlines function declared with the inline qualifier. • **1 - 8** - Inlines functions up to n levels deep. Level 0 is the same as `-inline on`. For `n`, enter 1 to 8 levels.
Auto Inline	Inlines small function even if they are not declared with the inline qualifier
Bottom-up Inlining	Check to control the bottom-up function inlining method. When active, the compiler inlines function code starting with the last function in the chain of functions calls, to the first one.

2.2.3.5 DSC Compiler > Processor

Use this panel to specify processor behavior. You can specify the file paths and define macros.

The following table lists and describes the compiler processor options for DSC.

Table 2-12. Tool Settings - DSC Compiler > Processor Options

Option	Description
Hardware DO Loops	Specifies the level of hardware DO loops: • **No DO Loops** - Compiler does not generate any • **No Nested DO Loops** - Compiler generates hardware DO loops, but does not nest them • **Nested DO Loops** - Compiler generates hardware DO loops, nesting them two deep. If hardware DO loops are enabled, debugging will be inconsistent about stepping into loops. Test immediately after this table contains additional Do-loop information.

Table continues on the next page...

Table 2-12. Tool Settings - DSC Compiler > Processor Options (continued)

Option	Description
Small Program Model	**Checked** - Compiler generates a more efficient switch table, provided that code fits into the range 0x0-0xFFFF. **Clear** - Compiler generates an ordinary switch table. Do not check this checkbox unless the entire program code fits into the 0x0-0xFFFF memory range.
Large Data Memory Model	**Checked** - Extends DSP56800E addressing range by providing 24-bit address capability to instructions. **Clear** - Does not extend address range. 24-bit address modes allow access beyond the 64K-byte boundary of 16-bit addressing.
Globals Live in Lower Memory	**Checked** - Compiler uses 24-bit addressing for pointer and stack operations, 16-bit addressing for access to global and static data. **Clear** - Compiler uses 24-bit addressing for all data access. This checkbox is available only if the Large Data Model checkbox is checked.
Zero-Initialized Globals LIve in Data Instead of BSS	**Checked** - Globals initialized to zero reside in the .data section. **Clear** - Globals initialized to zero reside in the .bss section.
Segregate Data Section	Check to segregate data section.
Pad Pipeline for Debugger	**Checked** - Mandatory for using the debugger. Inserts NOPs after certain branch instructions to make breakpoints work reliably. **Clear** - Does not insert such NOPs. If you select this option, you should select the same option in the assembler panel. Selecting this option increases code size by 5 percent. But not selecting this option risks nonrecovery after the debugger comes to breakpoint branch instructions.
Create Assembly Output	**Checked** - Assembler generates assembly code for each C file. **Clear** - Assembler does not generate assembly code for each C file. The pragma #asmoutput overrides this option for individual files.
Generate Code for Profiling	**Checked** - Compiler generates code for profiling. **Clear** - Compiler does not generate code for profiling.
Generates elf file for 56800EX core	**Checked** - Compiler generates elf file for 56800EX core. **Clear** - Compiler does not generate elf file for 56800EX core.
Check Inline Assembly for Pipeline	Specifies pipeline conflict detection during compiling of inline assembly source code: • **Not Detected** - compiler does not check for conflicts

Table continues on the next page...

Table 2-12. Tool Settings - DSC Compiler > Processor Options (continued)

Option	Description
	• **Conflict Error** - compiler issues error messages if it detects conflicts • **Conflict Error/Hardware Stall Warning** - compiler issues error messages if it detects conflicts, warnings if it detects hardware stalls
Check C Source for Pipeline	Specifies pipeline conflict detection during compiling of C source code: • **Not Detected** - compiler does not check for conflicts • **Conflict error** - compiler issues error messages if it detects conflicts

2.2.3.6 DSC Compiler > Language

Use this panel direct the DSC compiler to apply specific processing modes to the language source code. You can compile source files with just one collection at a time. To compile source files with multiple collections, you must compile the source code sequentially. After each compile iteration change the collection of settings that the DSC compiler uses.

The following table lists and describes the compiler optimization options for DSC.

Table 2-13. Tool Settings - DSC Compiler > Language Settings

Option	Description
ANSI Strict	Check to enable C compiler operate in strict ANSI mode. In this mode, the compiler strictly applies the rules of the ANSI/ISO specification to all input files. This setting is equivalent to specifying the `- ansi` command-line option. The compiler issues a warning for each ANSI/ISO extension it finds.
ANSI Keywords Only	Check to generate an error message for all non-standard keywords (ISO/IEC 9899-1990 C, §6.4.1). If you must write source code that strictly adheres to the ISO standard, enable this setting; is equivalent to pragma `only_std_keywords` and the command-line option `-stdkeywords`.
Enums Always Int	Check to use signed integers to represent enumerated constants and is equivalent to pragma `enumsalwaysint` and the command-line option `-enum`.
Use Unsigned Chars	Check to treat char declarations as unsigned char declarations and is equivalent to pragma `unsigned_char` and the command-line option `-char unsigned`.
Require Function Prototypes	Check to enforce the requirement of function prototypes. The compiler generates an error message if you define a previously referenced function that does not have a prototype. If you define the function before it is referenced but do not give it a prototype, this setting causes the compiler to issue a warning message.

Table continues on the next page...

Table 2-13. Tool Settings - DSC Compiler > Language Settings (continued)

Option	Description
Expand Trigraphs	Check to recognize trigraph sequences (ISO/IEC 9899-1990 C, §5.2.1.1); is equivalent to **pragma** `trigraphs` **and the command-line option** `-trigraphs`.
Legacy for-scoping	Check to generate an error message when the compiler encounters a variable scope usage that the ISO/IEC 14882:2003 C++ standard disallows, but is allowed in the C++ language specified in 'The Annotated C++ Reference Manual'.
Reuse Strings	Check to store only one copy of identical string literals and is equivalent to opposite of the pragma `dont_reuse_strings` and the command-line `option -string reuse`.
Pool Strings	Check to collect all string constants into a single data section in the object code it generates and is equivalent to pragma `pool_strings` and the command-line `option -strings pool`.
Other Flags	Specify additional command line options for the compiler; type in custom flags that are not otherwise available in the UI. **NOTE:** To enable CodeWarrior MCU V10.x to generate .lst file for each source file in DSC you need to specify -S in the **Other Flags** option.

2.2.4 DSC Assembler

Use this panel to specify the command, options, and expert settings for the build tool assembler. Additionally, the Assembler tree control includes the general and include file search path settings.

The following table lists and describes the compiler optimization options for DSC.

Table 2-14. Tool Settings - DSC Assembler

Option	Description
Command	Shows the location of the assembler executable file. You can specify additional command line options for the assembler; type in custom flags that are not otherwise available in the UI.
All options	Shows the actual command line the assembler will be called with.
Expert Settings Command line pattern	Shows the expert settings command line parameters; default is `${COMMAND} ${FLAGS} ${OUTPUT_FLAG} ${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}`.

2.2.4.1 DSC Assembler > Input

Use this panel to specify additional files the **DSC Assembler** should use. You can specify multiple additional libraries and library search paths. Also, you can change the order in which the IDE uses or searches the libraries.

The following table lists and describes the compiler optimization options for DSC.

Table 2-15. Tool Settings - DSC Assembler > Input

Option	Description
Prefix File	Specify a prefix file that you want the compiler to include at the top of each file.
Always Search User Paths (-nosyspath)	Performs a search of both the user and system paths, treating #include statements of the form #include <xyz> the same as the form #include "xyz".
User Path (-i)	Lets you add/update the user paths that the CodeWarrior IDE searches to find files in your project. You can add several kinds of paths including absolute and project-relative.
User Recursive Path (-ir)	Lets you add/update the recursive user paths that the CodeWarrior IDE searches to find files in your project. You can add several kinds of paths including absolute and project-relative.
System Path (-I- -I)	Lets you add/update the system paths that the CodeWarrior IDE searches to find files in your project. You can add several kinds of paths including absolute and project-relative.
System Recursive Path (-I- -ir)	Lets you add/update the recursive system paths that the CodeWarrior IDE searches to find files in your project. You can add several kinds of paths including absolute and project-relative.

2.2.4.2 DSC Assembler > General

Use this panel to specify additional files the **DSC Assembler** should use. You can specify multiple additional libraries and library search paths. Also, you can change the order in which the IDE uses or searches the libraries.

The following table lists and describes the assembler options for DSC.

Table 2-16. Tool Settings - DSC Assembler > General

Option	Description
Identifiers are Case Sensitive	Clear to instruct the assembler to ignore case in identifiers. By default, the option is checked.
Assert NOPs on Pipeline Conflicts	**Checked** - Assembler automatically resolves pipeline conflicts by inserting NOPs.

Table continues on the next page...

Table 2-16. Tool Settings - DSC Assembler > General (continued)

Option	Description
	Clear - Assembler does not insert NOPs; it reports pipeline conflicts in error messages. NOP is optional. The core will stall for you (delay the required time) even if you do not put the NOP.
Emit Warnings for NOP Assertions	**Checked** - Assembler issues a warning any time it inserts a NOP to prevent a pipeline conflict. **Clear** - Assembler does not issue such warnings. This checkbox is available only if the Assert NOPs on pipeline conflicts checkbox is checked.
Emit Warnings for Hardware Stalls	**Checked** - Assembler warns when a hardware stall occurs upon execution. **Clear** - Assembler does not issue such warnings. This option helps optimize the cycle count.
Pad Pipeline for Debugger	**Checked** - Mandatory for using the debugger. Inserts NOPs after certain branch instructions to make breakpoints work reliably. **Clear** - Does not insert such NOPs. If you select this option, you should select the same option in the processor settings panel. Selecting this option increases code size by 5 percent. But not selecting this option risks nonrecovery after the debugger comes to breakpoint branch instructions.
Emit Warnings for Odd SP Increment/Decrement	**Checked** - Enables assembler warnings about instructions that could misalign the stack frame. **Clear** - Does not enable such warnings.
Allow Legacy Instructions (default to 16-bit memory models)	**Checked** - Assembler permits legacy DSP56800 instruction syntax. **Clear** - Assembler does not permit this legacy syntax. Selecting this option sets the Default Data Memory Model and Default Program Memory Model values to 16 bits.
Generates elf file for 56800EX core	Check to generate elf file for 56800EX core that makes a program file out of the object files of your project.
Default Data Memory Model	Specifies 16- or 24-bits as the default size. Factory setting: 16 bits.
Default Program Memory Model	Specifies 16-, 19-, or 21-bits as the default size. Factory setting: 19 bits.
Other Flags	Specify additional command line options for the assembler; type in custom flags that are not otherwise available in the UI. **Note:** To enable CodeWarrior MCU V10.x to generate .lst file for each source file in DSC, you need to specify -S in the Other Flags option.

2.2.4.3 DSC Assembler > Output

Use this panel to control how the assembler generates the output file, as well as error and warning messages. You can specify whether to allocate constant objects in ROM, generate debugging information, and strip file path information.

The following table lists and describes the assembler output options for DSC.

Table 2-17. Tool Settings - DSC Assembler > Output

Option	Description
Generate Listing File	Instructs the assembler to generate a disassembly output file. The disassembly output file contains the file source, along with line numbers, relocation information, and macro expansion.
Expand Macros in Listing File	• **Checked** - Assembler macros expand in the assembler listing. • **Clear** - Assembler macros do not expand. This checkbox is available only if the Generate Listing File checkbox is checked.

2.2.5 DSC Preprocessor

Use this panel to specify the preprocessor settings for DSC.

The following table lists and describes the preprocessor options for DSC.

Table 2-18. Tool Settings - DSC Preprocesor

Option	Description
Command	Shows the location of the preprocessor executable file. You can specify additional command line options for the preprocessor; type in custom flags that are not otherwise available in the UI.
All options	Shows the actual command line the preprocessor will be called with.
Expert Settings Command line pattern	Shows the expert settings command line parameters; default is `${COMMAND} -E ${FLAGS} ${INPUTS}`.

2.2.5.1 DSC Preprocessor > Settings

Use this panel to specify the preprocessor settings of DSC Preprocessor.

The following table lists and describes the preprocessor settings options for DSC.

Table 2-19. Tool Settings - DSC Preprocessor > Settings

Option	Description
Emit File/Line Breaks	Check to notify file breaks (or #line breaks) appear in the output.
Emit #pragma directives	Check to show pragma directives in the preprocessor output. Essential for producing reproducible test cases for bug reports.
Emit #line Directives	Check to display file changes in comments (as before) or in #line directives.
Show Full Path	Check to control whether file changes show the full path or the base filename of the file.
Keep Comments	Check to display comments in the preprocessor output.
Keep Whitespace	Check to copy whitespaces in preprocessor output. This is useful for keeping the starting column aligned with the original source, though the compiler attempts to preserve space within the line. This does not apply when macros are expanded.

2.2.6 DSC Disassembler

Use this panel to specify the command, options, and expert settings for DSC Disassembler.

The following table lists and describes the disassembler options for DSC.

Table 2-20. Tool Settings - DSC Disassembler

Option	Description
Command	Shows the location of the disassembler executable file. Default value is `""${DSC_ToolsDir}/mwld56800e"`. You can specify additional command line options for the disassembler; type in custom flags that are not otherwise available in the UI.
All options	Shows the actual command line the disassembler will be called with.
Expert settings Command line pattern	Shows the expert settings command line parameters; default is `${COMMAND} -dis ${FLAGS} ${INPUTS}`

2.2.6.1 DSC Disassembler > Settings

Use this panel to specify disassembler settings.

The following table lists and describes the disassembler settings options for DSC.

Table 2-21. Tool Settings - DSC Disassembler > Settings

Option	Description
Show Headers	Check to display headers in the listing file; disassembler writes listing headers, titles, and subtitles to the listing file
Show Symbol and String Tables	Check to display symbol and string tables directives to the listing file
Verbose Information	Tells the compiler to provide verbose, cumulative information in messages.
Show Relocations	Check to have the disassembler show information about relocated symbols. Clear to prevent the disassembler from showing information about relocated symbols.
Show Code Modules	Check to show core modules in the listing file
Show Extended Mnemonics	Check to show the extended mnemonics in the listing file
Show Addresses and Opcodes	Check to show the addresses and object code in the listing file
Show Source Code	Check to show the source code in the listing file
Show Comments	Check to show the comments in the listing file
Show Data Modules	Check to show the data modules in the listing file
Show Exception Tables	Check to disassemble exception tables in the listing file
Show Debug Information	Check to generate symbolic information for debugging the build target

Chapter 3
Using Build Tools on the Command Line

The CodeWarrior command line compilers and assemblers translate source code (for example, C and C++) into object code, storing this object in files. CodeWarrior command-line linkers then combine one or more of these object code files to produce an executable image ready to load and execute on the target platform.

Each command-line tool has options that you configure when you invoke the tool.

The CodeWarrior IDE (Integrated Development Environment) uses these same compilers and linkers, however Freescale provides versions of these tools that you can directly invoke on the command line.

This chapter contains these topics:

3.1 Naming Conventions

The names of the CodeWarrior command-line tools follow a convention:

```
mwtoolplatform
```

where *tool* is `cc` for the C/C++ compiler, `ld` for the linker, and `asm` for the assembler.

platform is usually the target platform that the tool generates software for, except where there are multiple versions of tools for a target platform.

For example, the command-line compiler, assembler, and linker for the dsp56800 are named `mwcc56800`, `mwasm56800`, and `mwld56800`, respectively; and for the dsp56800e are named `mwcc56800e`, `mwasm56800e`, and `mwld56800e`, respectively.

3.2 Configuring Command-Line Tools

To use the command-line tools, several environment variables must be changed or defined.

If you are using CodeWarrior command-line tools with Microsoft Windows, environment variables may be assigned in **Environment variables** under **System Properties** of control panel.

The CodeWarrior command-line tools refer to environment variables for configuration information:

- CWFolder Environment Variable
- Setting the PATH Environment Variable

3.2.1 CWFolder Environment Variable

In this example, `%CWFolder%` refers to the path where CodeWarrior for 56800 was installed. Note that it is not necessary to include quote marks when defining environment variables that include spaces. Windows does not strip out the quotes and this leads to unknown directory warnings. Use the following syntax if defining variables in batch files or at the command line.

```
set CWFolder=C:\Freescale\CW MCU v10.x\MCU
```

```
set PATH=%PATH%;%CWFolder%\DSP56800x_EABI_Tools\command_line_tools
```

3.2.2 Setting the PATH Environment Variable

The PATH variable should include the paths for the 56800/E tools as shown in the following listing.

Listing: Example of PATH Settings

```
%CWFolder%\MCU\DSP56800x_EABI_Tools\command_line_tools
```

In order for FlexLM to work properly, you can simply copy the following file into the directory from which you will be using the command line tools:

```
..\CW MCU v10.x\MCU\license.dat
```

Alternately, you can define the variable LM_LICENSE_FILE as:

```
%CWFolder%\MCU\license.dat
```

This variable points to license information. It may point to alternate versions of this file, as needed.

3.3 Invoking Command-Line Tools

To compile, assemble, link, or perform some other programming task with the CodeWarrior command-line tools, you type a command at a command line's prompt. This command specifies the tool you want to run, what options to use while the tool runs, and what files the tool should operate on.

The form of a command to run a command-line tool is

```
tool options files
```

where *tool* is the name of the CodeWarrior command-line tool to invoke, *options* is a list of zero or more options that specify to the tool what operation it should perform and how it should be performed, and files is a list of files zero or more files that the tool should operate on.

Which options and files you should specify depend on what operation you want the tool to perform.

The tool then performs the operation on the files you specify. If the tool is successful it simply finishes its operation and a new prompt appears at the command line. If the tool encounters problems it reports these problems as text messages on the command-line before a new prompt appears.

Scripts that automate the process to build a piece of software contain commands to invoke command-line tools. For example, the `make` tool, a common software development tool, uses scripts to manage dependencies among source code files and invoke command-line compilers, assemblers and linkers as needed, much like the CodeWarrior IDE's project manager.

3.4 Getting Help

To show short descriptions of a tool's options, type this command at the command line:

```
tool -help
```

where *tool* is the name of the CodeWarrior build tool.

To show only a few lines of help information at a time, pipe the tool's output to a pager program. For example,

```
tool -help | more
```

will use the `more` pager program to display the help information.

3.4.1 Help Guidelines

Enter the following command in a **Command Prompt** window to see a list of specifications that describe how options are formatted:

```
tool -help usage
```

3.4.1.1 Parameter Formats

Parameters in an option are formatted as follows:

- A parameter included in brackets " `[]`" is optional.
- Use of the ellipsis " `...`" character indicates that the previous type of parameter may be repeated as a list.

3.4.1.2 Option Formats

Options are formatted as follows:

- For most options, the option and the parameters are separated by a space as in " `-xxx param`". When the option's name is " `-xxx+`", however, the parameter must directly follow the option, without the " `+`" character (as in " `-xxx45`") and with no space separator.
- An option given as " `-[no]xxx`" may be issued as " `-xxx`" or " `-noxxx`". The use of " `-noxxx`" reverses the meaning of the option.
- When an option is specified as " `-xxx | yy[y] | zzz`", then either " `-xxx`", " `-yy`", " `-yyy`", or " `-zzz`" matches the option.
- The symbols "," and "=" separate options and parameters unconditionally; to include one of these symbols in a parameter or filename, escape it (e.g., as " `\,`" in `mwcc file.c \,v`).

3.4.1.3 Common Terms

These common terms appear in many option descriptions:

- A "cased" option is considered case-sensitive. By default, no options are case-sensitive.
- "compatibility" indicates that the option is borrowed from another vendor's tool and its behavior may only approximate its counterpart.
- A "global" option has an effect over the entire command line and is parsed before any other options. When several global options are specified, they are interpreted in order.
- A "deprecated" option will be eliminated in the future and should no longer be used. An alternative form is supplied.
- An "ignored" option is accepted by the tool but has no effect.
- A "meaningless" option is accepted by the tool but probably has no meaning for the target OS.
- An "obsolete" option indicates a deprecated option that is no longer available.
- A "substituted" option has the same effect as another option. This points out a preferred form and prevents confusion when similar options appear in the help.
- Use of "default" in the help text indicates that the given value or variation of an option is used unless otherwise overridden.

This tool calls the linker (unless a compiler option such as `-c` prevents it) and understands linker options - use '`-help tool=other`' to see them. Options marked "passed to linker" are used by the compiler and the linker; options marked "for linker" are used only by the linker. When using the compiler and linker separately, you must pass the common options to both.

3.5 File Name Extensions

Files specified on the command line are identified by contents and file extension, as in the CodeWarrior IDE.

The command-line version of the CodeWarrior C/C++ compiler accepts non-standard file extensions as source but also emits a warning. By default, the compiler assumes that a file with any extensions besides `.c`, `.h`, `.pch` is C++ source. The linker ignores all files that it can not identify as object code, libraries, or command files.

Linker command files must end in `.cmd`. They may be simply added to the link line, for example, for 56800:

```
mwld56800e file.o "MSL C 56800E.lib" "Runtime 56800E.Lib" linker.cmd
```

For more information on linker command files, refer to the *Targeting* manual for your platform.

3.6 Specifying Source File Locations

Several environment variables are used at build time to search for system include paths and libraries which can shorten command lines for many tasks. All of the variables mentioned here are lists which are separated by semicolons (";") in Windows and colons (":") in Solaris.

For example, in 56800, unless `-nodefaults` is passed to on the command line, the compiler searches for an environment variable called `MWC56800Includes` for the DSP56800 and `MWC56800EIncludes` for the DSP56800E. This variable contains a list of system access paths to be searched after the system access paths specified by the user. The assembler also does this, using the variable `MWAsm56800Includes` for the DSP56800 and `MWAsm56800EIncludes` for the DSP56800E.

Analogously, unless `-nodefaults` or `-disassemble` is given, the linker will search the environment for a list of system access paths and system library files to be added to the end of the search and link orders. For example, with 56800, the variable `MW56800Libraries` and `MW56800ELibraries` contains a list of system library paths to search for files, libraries, and command files.

Associated with this list is the variable `MW56800LibraryFiles` and `MW56800ELibraryFiles` which contains a list of libraries (or object files or command files) to add to the end of the link order. These files may be located in any of the cumulative access paths at runtime.

3.7 Environmental Variables

There are environmental variables for the DSP56800 and DSP56800E.

The environmental variables for the DSP56800 are:

- MW56800Libraries: a semicolon separated list of paths to the libraries
- MW56800LibraryFiles: a semicolon separated list of libraries to be linked against
- MWAsm56800Includes: a semicolon separated list of paths to files needed by the assembler
- MWC56800Includes: a semicolon separated list of paths to files needed by the assembler

The environmental variables for the DSP56800E are:

- MW56800ELibraries: a semicolon separated list of paths to the libraries
- MW56800ELibraryFiles: a semicolon separated list of libraries to be linked against
- MWAsm56800EIncludes: a semicolon separated list of paths to files needed by the assembler
- MWC56800EIncludes: a semicolon separated list of paths to files needed by the assembler

3.8 Standard C and C++ Conformance Options

The Standard C and C++ Conformance options are:

- -ansi
- -stdkeywords
- -strict

3.8.1 -ansi

Controls the ANSI conformance options, overriding the given settings.

Syntax

```
-ansi keyword
```

The arguments for `keyword` are:

`off`

Turn ANSI conformance off. Same as `-stdkeywordsoff`, `-enummin`, and `-strictoff`.

`on | relaxed`

Turn ANSI conformance on in relaxed mode. Same as `-stdkeywordson`, `-enummin`, and `-stricton`.

`strict`

Turn ANSI conformance on in strict mode. Same as `-stdkeywordson`, `-enumint`, and `-stricton`.

3.8.2 -stdkeywords

Controls the requirement for the use of ANSI standard keywords.

Syntax

```
-stdkeywords on | off
```

Remarks

Default setting is `off`.

3.8.3 -strict

Controls the use of non-standard ANSI language features.

Syntax

```
-strict on | off
```

Remarks

Default setting is `off`.

3.9 Language Translation and Extensions Options

The Language Translation and Extensions options are:

- -char
- -defaults
- -encoding
- -flag
- -fullLicenseSearch
- -gccext
- -gcc_extensions
- -M
- -make
- -mapcr
- -MM
- -MD
- -MMD
- -Mfile
- -MMfile
- -MDfile
- -MMDfile
- -multibyteaware
- -nolonglong
- -once
- -pragma
- -relax_pointers
- -requireprotos
- -search
- -trigraphs

3.9.1 -char

Controls the default sign of the `char` data type.

Syntax

```
-char keyword
```

The arguments for `keyword` are:

`signed`

char data items are signed.

`unsigned`

char data items are unsigned.

Remarks

The default is `signed`.

3.9.2 -defaults

Controls whether the compiler uses additional environment variables to provide default settings.

Syntax

```
-defaults
-[no]defaults
```

Remarks

This command is global. To enable the command-line compiler to use the same set of default settings as the CodeWarrior IDE, use `-defaults`. For example, in the IDE, all access paths and libraries are explicit. `defaults` is the default setting.

Use `-nodefaults` to disable the use of additional environment variables.

3.9.3 -encoding

Specify the default source encoding used by the compiler.

Syntax

```
-enc[oding] keyword
```

The options for `keyword` are:

`ascii`

American Standard Code for Information Interchange (ASCII) format. This is the default.

`autodetect | multibyte | mb`

Scan file for multibyte encoding.

`system`

Use local system format.

`UTF[8 | -8]`

Unicode Transformation Format (UTF).

`SJIS | Shift-JIS | ShiftJIS`

Shift Japanese Industrial Standard (Shift-JIS) format.

`EUC[JP | -JP]`

Japanese Extended UNIX Code (EUCJP) format.

`ISO[2022JP | -2022-JP]`

International Organization of Standards (ISO) Japanese format.

Remarks

The compiler automatically detects UTF-8 (Unicode Transformation Format) header or UCS-2/UCS-4 (Uniform Communications Standard) encodings regardless of setting. The default setting is `ascii`.

3.9.4 -flag

Specify compiler `#pragma` as either `on` or `off`.

Syntax

```
-fl[ag] [no-]pragma
```

Examples

```
-flag foo
```

is equivalent to `#pragma foo on`.

```
-flag no-foo
```

is the same as `#pragma foo off`.

3.9.5 -fullLicenseSearch

Uses more robust search for valid license files.

Syntax

```
-fullLicenseSearch
```

Remarks

This command is global. It might result in somewhat longer builds.

3.9.6 -gccext

Enable GCC (Gnu Compiler Collection) C language extensions.

Syntax

```
-gcc[ext] on | off
```

Remarks

The default setting is `off`.

3.9.7 -gcc_extensions

Equivalent to the `-gccext` option.

Syntax

```
-gcc[_extensions] on | off
```

3.9.8 -M

Scan source files for dependencies and emit a Makefile, without generating object code.

Syntax

```
-M
```

Remarks

This command is global and case-sensitive.

3.9.9 -make

Scan source files for dependencies and emit a Makefile, without generating object code.

Syntax

```
-make
```

Remarks

This command is global.

3.9.10 -mapcr

Swaps the values of the `\n` and `\r` escape characters.

Syntax

```
-mapcr
```

```
-nomapcr
```

Remarks

The `-mapcr` option tells the compiler to treat the `'\n'` character as ASCII 13 and the `'\r'` character as ASCII 10. The `-nomapcr` option tells the compiler to treat these characters as ASCII 10 and 13, respectively.

3.9.11 -MM

Scan source files for dependencies and emit a Makefile, without generating object code or listing system `#include` files.

Syntax

```
-MM
```

Remarks

This command is global and case-sensitive.

3.9.12 -MD

Scan source files for dependencies and emit a Makefile, generate object code, and write a dependency map.

Syntax

```
-MD
```

Remarks

This command is global and case-sensitive.

3.9.13 -MMD

Scan source files for dependencies and emit a Makefile, generate object code, write a dependency map, without listing system `#include` files.

Syntax

```
-MMD
```

Remarks

This command is global and case-sensitive.

3.9.14 -Mfile

Scans source files for dependencies and emit Makefile, does not generate object code, writes a dependency map to the specified file.

Syntax

```
-Mfile file
```

Remarks

This command is global and case-sensitive.

3.9.15 -MMfile

Scans source files for dependencies and emit a Makefile, without generating object code or listing system `#include` files, and writes a dependency map to the specified file.

Syntax

```
-MMfile file
```

Remarks

This command is global and case-sensitive.

3.9.16 -MDfile

Scans source files for dependencies and emit a Makefile, generates object code, and writes a dependency map to the specified file.

Syntax

```
-MDfile file
```

Remarks

This command is global and case-sensitive.

3.9.17 -MMDfile

Scans source files for dependencies and emit a Makefile, generates object code, writes a dependency map to the specified file, without listing system `#include` files.

Syntax

```
-MMDfile file
```

Remarks

This command is global and case-sensitive.

3.9.18 -multibyteaware

Allows multi-byte characters encodings in source text.

Syntax

```
-multibyte[aware]
-nomultibyte[aware]
```

3.9.19 -nolonglong

Disables `long long` support..

Syntax

```
-nolonglong
```

3.9.20 -once

Prevents header files from being processed more than once.

Syntax

```
-once
```

Remarks

You can also add #pragma once on in a prefix file.

3.9.21 -pragma

Defines a pragma for the compiler.

Syntax

```
-pragma 'name ["setting"]'
```

The arguments are:

`name`

Name of the new pragma enclosed in single-quotes.

`setting`

Setting for the new pragma. When adding a setting, setting must be enclosed in double-quotes.

3.9.22 -relax_pointers

Relaxes the pointer type-checking rules in C.

Syntax

```
-relaxpointers
```

Remarks

This option is equivalent to

```
#pragma mpwc_relax on
```

3.9.23 -requireprotos

Controls whether or not the compiler should expect function prototypes.

Syntax

```
-r[equireprotos]
```

3.9.24 -search

Globally searches across paths for source files, object code, and libraries specified in the command line.

Syntax

```
-search
```

3.9.25 -trigraphs

Controls the use of ISO trigraph sequences.

Syntax

```
-trigraphs on | off
```

Remarks

Default setting is `off`.

3.10 Errors, Warnings, and Diagnostic Options

The Errors, Warnings, and Diagnostic options are:

- -disassemble
- -help
- -maxerrors
- -maxwarnings
- -msgstyle
- -nofail
- -progress
- -S
- -stderr
- -verbose
- -version
- -timing
- -warnings
- -wraplines

3.10.1 -disassemble

Tells the command-line tool to disassemble files and send result to `stdout`.

Syntax

```
-dis[assemble]
```

Remarks

This option is global.

3.10.2 -help

Lists descriptions of the CodeWarrior tool's command-line options.

Syntax

```
-help [keyword [,...]]
```

The options for `keyword` are:

`all`

Show all standard options

`group=keyword`

Show help for groups whose names contain 'keyword' (case-sensitive); for 'keyword', maximum length 63 chars

`[no]compatible`

Use `compatible` to show options compatible with this compiler. Use `nocompatible` to show options that do not work with this compiler.

`[no]deprecated`

Show deprecated options

`[no]ignored`

Show ignored options

`[no]meaningless`

Show options meaningless for this target

`[no]normal`

Show only standard options

`[no]obsolete`

Show obsolete options

`[no]spaces`

Insert blank lines between options in printout.

`opt[ion]=name`

Show help for a given option; for 'name', maximum length 63 chars

`search=keyword`

Show help for an option whose name or help contains 'keyword' (case-sensitive); for 'keyword', maximum length 63 chars

`tool=keyword[ all | this | other|skipped | both ]`

Categorize groups of options by tool; default.

- `all`-show all options available in this tool
- `this`-show options executed by this tool; default
- `other|skipped`-show options passed to another tool
- `both`-show options used in all tools

`usage`

Displays usage information.

3.10.3 -maxerrors

Specify the maximum number of errors to show.

Syntax

```
-maxerrors max
```

`max`

Use `max` to specify the number of errors. Common values are:

- `0` (zero) - disable maximum count, show all errors.
- `100` - Default setting.

3.10.4 -maxwarnings

Specify the maximum number of warnings to show.

Syntax

```
-maxwarnings max
```

`max`

Use `max` to specify the number of warnings. Common values are:

- `0` (zero) - Disable maximum count (default).
- `n` - Maximum number of warnings to show.

3.10.5 -msgstyle

Controls the style used to show error and warning messages.

Syntax

```
-msgstyle keyword
```

The options for `keyword` are:

`EnterpriseIDE`

Uses Enterprise IDE message style.

`gcc`

Uses gcc message style.

`ide`

Uses CodeWarrior's Integrated Development Environment (IDE) message style.

`mpw`

Uses Macintosh Programmer's Workshop (MPW®) message style.

`parseable`

Uses context-free machine parseable message style.

`std`

Uses standard message style. This is the default.

3.10.6 -nofail

Continue processing after getting errors in earlier files.

Syntax

```
-nofail
```

3.10.7 -progress

Show progress and version information.

Syntax

```
-progress
```

3.10.8 -S

Disassemble all files and send output to a file. This command is global and case-sensitive.

Syntax

```
-S
```

3.10.9 -stderr

Use the standard error stream to report error and warning messages.

Syntax

```
-stderr
```

```
-nostderr
```

Remarks

The `-stderr` option specifies to the compiler, and other tools that it invokes, that error and warning messages should be sent to the standard error stream.

The `-nostderr` option specifies that error and warning messages should be sent to the standard output stream.

3.10.10 -verbose

Tells the compiler to provide verbose, cumulative information in messages.

Syntax

```
-v[erbose]
```

Remarks

Use of this argument implies the use of the The -progress argument.

3.10.11 -version

Displays version, configuration, and build data.

Syntax

```
-v[ersion]
```

3.10.12 -timing

Shows the amount of time that the tool used to perform an action.

Syntax

```
-timing
```

3.10.13 -warnings

Specify which warnings the command-line tool issues. This command is global.

Syntax

```
-w[arning] keyword [,...]
```

The options for `keyword` are:

`off`

Turn off all warnings. Passed to all tools. Prefix file setting: `#pragma warning off`.

`on`

Turn on most warnings. Passed to all tools. Prefix file setting: `#pragma warning on`.

`[no]cmdline`

Issue command-line driver/parser warnings

`[no]err[or] | [no]iserr[or]`

Treat warnings as errors. Passed to all tools. Prefix file setting: `#pragma warning_errors`.

`most`

Turn on most warnings.

`all`

Turn on all warnings and require prototypes.

`full`

Turn on all warnings including spurious warnings and require prototypes.

`[no]pragmas | [no]illpragmas`

Issue warnings on illegal #pragmas. Prefix file setting: `#pragma warn_illpragma`.

`[no]empty[decl]`

Issue warnings on empty declarations. Prefix file setting: `#pragma warn_emptydelc`.

`[no]possible | [no]unwanted`

Issue warnings on possible unwanted effects. Prefix file setting: `#pragma warn_possunwanted`.

`[no]unusedarg`

Issue warnings on unused arguments. Prefix file setting: `#pragma warn_unusedarg`.

`[no]unusedvar`

Issue warnings on unused variables. Prefix file setting: `#pragma warn_unusedvar`.

`[no]unused`

Same as `-w [no]unusedarg,[no]unusedvar`.

`[no]extracomma | [no]comma`

Issue warnings on extra commas in enumerations. Prefix file setting: `#pragma warn_extracomma`.

`[no]pedantic | [no]extended`

pedantic error checking

`[no]hidevirtual | [no]hidden[virtual]`

Issue warnings on hidden virtual functions. Prefix file setting: `#pragma warn_hidevirtual`.

`[no]implicit[conv]`

Issue warnings on implicit arithmetic conversions. Implies `-warn impl_float2int,impl_signedunsigned`.

`[no]impl_int2float`

Issue warnings on implicit integral to floating conversions. Prefix file setting: `#pragma warn_impl_i2f_conv`.

`[no]impl_float2int`

Issue warnings on implicit floating to integral conversions. Prefix file setting: `#pragma warn_impl_f2i_conv`.

`[no]impl_signedunsigned`

Issue warnings on implicit signed/unsigned conversions.

`[no]notinlined`

Issue warning when `inline` functions are not inlined. Prefix file setting: `#pragma warn_notinlined`.

`[no]largeargs`

Issue warning when passing large arguments to unprototyped functions. Prefix file setting: `#pragma warn_largeargs`.

`[no]structclass`

Issue warning on inconsistent use of `class` and `struct`. Prefix file setting: `#pragma warn_structclass`.

`[no]padding`

Issue warning when padding is added between `struct` members. Prefix file setting: `#pragma warn_padding`.

`[no]notused`

Issue warning when the result of non-void-returning functions are not used. Prefix file setting: `#pragma warn_resultnotused`.

`[no]missingreturn`

Issue warning when a return without a value in non-void-returning function occurs. Prefix file setting: `#pragma warn_missingreturn`.

`[no]unusedexpr`

Issue warning when encountering the use of expressions as statements without side effects. Prefix file setting: `#pragma warn_no_side_effect`.

`[no]ptrintconv`

Issue warning when loss conversions occur from pointers to integers.

`[no]anyptrintconv`

Issue warning on any conversion of pointers to integers. Prefix file setting: `#pragma warn_ptr_int_conv`.

`[no]undef[macro]`

Issue warning on the use of undefined macros in `#if/#elif` conditionals. Prefix file setting: `#pragma warn_undefmacro`.

`[no]filecaps`

Issue warning when `#include "..."` statements use incorrect capitalization. Prefix file setting: `#pragma warn_filenamecaps`.

`[no]sysfilecaps`

Issue warning when `#include <...>` statements use incorrect capitalization. Prefix file setting: `#pragma warn_filenamecaps_system`.

`[no]tokenpasting`

Issue warning when token is not formed by `##` operator. Prefix file setting: `#pragma warn_illtokenpasting`.

`[no]relax_i2i_conv`

Issue relax warnings for implicit integer to integer arithmetic conversions (off for full, on otherwise).

`[no]alias_ptr_conv`

Generate warnings for potentially dangerous pointer casts (full).

`display | dump`

Display list of active warnings.

Description

Choose **Edit > targetname Settings** from the CodeWarrior IDE's menu bar, then select the **C/C++ Warnings** settings panel. Enable or disable specific warnings by clicking the appropriate checkboxes.

3.10.14 -wraplines

Controls the word wrapping of messages.

Syntax

```
-wraplines

-nowraplines
```

3.11 Preprocessing and Precompilation Options

The Preprocessing and Precompilation options are:

- -allow_macro_redefs
- -convertpaths
- -cwd
- -D+
- -define
- -E
- -EP
- -gccdepends
- -gccincludes
- -I-
- -I+
- -include
- -ir
- -noprecompile
- -nosyspath
- -P
- -precompile
- -preprocess
- -ppopt
- -prefix
- -stdinc
- -U+
- -undefine

3.11.1 -allow_macro_redefs

Allows macro redefinitions without errors or warnings.

Syntax

```
-allow_macro_redefs
```

3.11.2 -convertpaths

Instructs the compiler to interpret `#include` file paths specified for a foreign operating system. This command is global.

Syntax

```
-[no]convertpaths
```

Remarks

The CodeWarrior compiler can interpret file paths from several different operating systems. Each operating system uses unique characters as path separators. These separators include:

- Mac OS® - colon " `:`" (`:sys:stat.h`)
- UNIX - forward slash " `/`" (`sys/stat.h`)
- Windows® - backward slash " `\`" (`sys\stat.h`)

When `convertpaths` is enabled, the compiler can correctly interpret and use paths like `<sys/stat.h>` or `<:sys:stat.h>`. However, when enabled, (`/`) and (`:`) separate directories and cannot be used in filenames.

NOTE

This is not a problem on Windows since these characters are already disallowed in file names. It is safe to leave this option on.

When `noconvertpaths` is enabled, the compiler can only interpret paths that use the Windows form, like `<\sys\stat.h>`.

3.11.3 -cwd

Controls where a search begins for `#include` files. The path represented by *keyword* is searched before searching access paths defined for the build target.

Syntax

```
-cwd keyword
```

The options for `keyword` are:

`explicit`

No implicit directory. Search `-I` or `-ir` paths.

`include`

Begin search in directory of referencing file.

`proj`

Begin search in current working directory (default).

`source`

Begin search in directory that contains the source file.

3.11.4 -D+

Same as the `-define` option.

Syntax

```
-D+name
```

The parameters are:

`name`

The symbol name to define. Symbol is set to 1.

3.11.5 -define

Defines a preprocessor symbol.

Syntax

```
-d[efine]name[=value]
```

The parameters are:

`name`

The symbol name to define.

`value`

The value to assign to symbol name. If no value is specified, set symbol value equal to 1.

3.11.6 -E

Tells the command-line tool to preprocess source files. This command is global and case-sensitive.

Syntax

```
-E
```

3.11.7 -EP

Tells the command-line tool to preprocess source files that are stripped of `#line` directives. This command is global and case-sensitive.

Syntax

```
-EP
```

Remarks

Output is generated using the `#pragma simple_predump on` setting and sent to a new unsaved editor window.

3.11.8 -gccdepends

Writes dependency file (-MD, -MMD) with name and location based on output file, which is compatible with gcc 3.x, else writes to the current directory with filename based on the source file.

Syntax

```
-[no]gccdep[ends]
```

Remarks

This command is global.

3.11.9 -gccincludes

Controls the compilers use of GCC `#include` semantics.

Syntax

```
-gccinc[ludes]
```

Remarks

Use `-gccinclude` to control the CodeWarrior compiler understanding of GCC semantics. When enabled, the semantics include:

- Adds `-I-` paths to the systems list if `-I-` is not already specified
- Search referencing file's directory first for `#include` files (same as `-cwd include`) The compiler and IDE only search access paths, and do not take the currently `#include` file into account.

This command is global.

3.11.10 -I-

Changes the build target's search order of access paths to start with the system paths list. This command is global.

Syntax

```
-I-

-i-
```

Remarks

The compiler can search `#include` files in several different ways. Use `-I-` to set the search order as follows:

- For include statements of the form `#include"xyz"`, the compiler first searches user paths, then the system paths
- For include statements of the form `#include<xyz>`, the compiler searches only system paths

3.11.11 -I+

Appends a non-recursive access path to the current `#include` list. This command is global and case-sensitive.

Syntax

```
-I+path
```

```
-i path
```

The parameters are:

`path`

The non-recursive access path to append.

3.11.12 -include

Defines the name of the text file or precompiled header file to add to every source file processed.

Syntax

```
  -include file
```

`file`

Name of text file or precompiled header file to prefix to all source files.

Remarks

With the command line tool, you can add multiple prefix files all of which are included in a meta-prefix file.

3.11.13 -ir

Appends a recursive access path to the current `#include` list. This command is global.

Syntax

```
-ir path
```

The parameters are:

`path`

The recursive access path to append.

3.11.14 -noprecompile

Do not precompile any source files based upon the filename extension.

Syntax

```
-noprecompile
```

3.11.15 -nosyspath

Perform searches of both the user and system paths, treating `#include` statements of the form `#include<xyz>` the same as the form `#include"xyz"`.

Syntax

```
-nosyspath
```

Remarks

This command is global.

3.11.16 -P

Preprocess the source files without generating object code, and send output to file. This command is global and case-sensitive.

Syntax

```
-P
```

3.11.17 -precompile

Precompile a header file from selected source files.

Syntax

```
-precompile file | dir | ""
```

The parameters are:

`file`

If specified, the precompiled header name.

`dir`

If specified, the directory to store the header file.

`""`

If "" is specified, write header file to location specified in source code. If neither argument is specified, the header file name is derived from the source file name.

Remarks

The driver determines whether to precompile a file based on its extension. The statement `-precompile filesource is equivalent to -c -o filesource`.

3.11.18 -preprocess

Preprocess the source files. This command is global.

Syntax

```
-preprocess
```

3.11.19 -ppopt

Specify options affecting the preprocessed output. The default settings is `break`.

Syntax

```
-ppopt keyword [,...]
```

The arguments for `keyword` are:

`[no]break`

Emit file and line breaks. This is the default.

`[no]line`

Controls whether #line directives are emitted or just comments. The default is `line`.

`[no]full[path]`

Controls whether full paths are emitted or just the base filename. The default is `fullpath`.

`[no]pragma`

Controls whether #pragma directives are kept or stripped. The default is `pragma`.

`[no]comment`

Controls whether comments are kept or stripped.

`[no]space`

Controls whether whitespace is kept or stripped. The default is `space`.

3.11.20 -prefix

Add contents a text file or precompiled header as a prefix to all source files.

Syntax

```
-prefix file
```

3.11.21 -stdinc

Use standard system include paths as specified by the environment variable `%MWCIncludes%`.

Syntax

```
-stdinc
-nostdinc
```

Remarks

Add this option after all system `-I` paths.

3.11.22 -U+

Same as the `-undefine` option.

Syntax

```
-U+name
```

3.11.23 -undefine

Undefine the specified symbol name. This command is case-sensitive.

Syntax

```
-u[ndefine] name
-U+name
```

The parameters are:

`name`

The symbol name to undefine.

3.12 Library and Linking Options

The Library and Linking options are:

- -keepobjects
- -map showbyte
- -nolink
- -o

3.12.1 -keepobjects

Retains or deletes object files after invoking the linker.

Syntax

```
-keepobj[ects]
-nokeepobj[ects]
```

Remarks

Use `-keepobjects` to retain object files after invoking the linker. Use - `nokeepobjects` to delete object files after linking. This command is global.

NOTE

Object files are always kept when compiling.

3.12.2 -map showbyte

This option activates the *Annotate Byte Symbols* IDE feature.

3.12.3 -nolink

Compile the source files, without linking.

Syntax

```
-nolink
```

Remarks

This command is global.

3.12.4 -o

Specify the output filename or directory for storing object files or text output during compilation, or the output file if calling the linker.

Syntax

```
-o file | dir
```

The parameters are:

`file`

The output file name.

`dir`

The directory to store object files or text output.

Remarks

Choose **Edit > targetname Settings** from the CodeWarrior IDE's menu bar, then select the **Access Paths** settings panel. Enable the **Always Search User Paths** option.

3.13 Object Code Organization and Generation Options

The Object Code Organization and Generation options are:

- -allowREP
- -asmout
- -c
- -chkasm
- -chkcsrcpipeline
- -codegen
- -constarray
- -Do
- -enum
- -ext
- -for_scoping
- -globalsInLowerMemory
- -hprog | -hugeprog
- -initializedzerodata
- -ldata | -largedata
- -largeAddrInSdm
- -min_enum_size
- -padpipe
- -profile
- -scheduling
- -segchardata
- -sprog | -smallprog
- -stackseq
- -strings
- -swp
- -V3

3.13.1 -allowREP

Generates REP instructions.

Syntax

```
-allowREP
-noallowREP
```

3.13.2 -asmout

Produces assembly file output.

Syntax

```
-asmout
-noasmout
```

3.13.3 -c

Instructs the compiler to compile but not link the object code.

Syntax

```
-c
```

Remarks

This option is global.

3.13.4 -chkasm

Check for pipeline in inline assembly sources.

Syntax

```
-chkasm keyword
```

The arguments of `keyword` are:

`off`

No check for pipeline in inline assembly sources. This is the default.

`conflict`

To display error on pipeline conflict.

`conflict_and_stall`

To display error on pipeline conflict and warning on hardware stalls.

3.13.5 -chkcsrcpipeline

Checks for pipeline in C sources.

Syntax

```
-chkcsrpipeline keyword
```

The arguments of `keyword` are:

`off`

No check for pipeline in C sources. This is the default.

`conflict`

To display error on pipeline conflict.

3.13.6 -codegen

Controls the generation of object code.

Syntax

```
-codegen
-nocodegen
```

Remarks

This option is global.

3.13.7 -constarray

Applies constant to array optimization.

Syntax

```
-constarray
-noconstarray
```

3.13.8 -Do

Specifies hardware DO loops.

Syntax

```
-Do keyword
```

The arguments of `keyword` are:

`off`

Does not specify hardware DO loops. This is the default.

`nonested`

Specifies hardware DO loops but not the nested ones.

`nested`

Specifies nested hardware DO loops.

3.13.9 -enum

Specify the default size for enumeration types. Default setting is `min`.

Syntax

```
-enum keyword
```

The arguments for `keyword` are:

`int`

Use `int` size for enumerated types.

`min`

Use minimum size for enumerated types. This is the default.

3.13.10 -ext

Tells the command-line tool the extension to apply to object files.

Syntax

```
-ext extension
```

The value of `extension` is:

`extension`

The extension to apply to object files. Use these rules to specify the extension:

- Limited to a maximum length of 14-characters
- Extensions specified without a leading period (`extension`) replace the source file's extension. For example, if `extension == o`, then `source.cpp` becomes `source.o`.
- Extensions specified with a leading period (`.extension`) are appended to the object files name. For example, if `extension == .o`, then `source.cpp` becomes `source.cpp.o`.

Remarks

This command is global. The default setting is no extension.

3.13.11 -for_scoping

Controls legacy (non-standard) for-scoping behavior.

Syntax

```
-for_scoping on|off
```

Remarks

If enabled, variables declared in `for` loops are visible to the enclosing scope. If disabled, such variables are scoped to the loop only. The default value is `off`.

3.13.12 -globalsInLowerMemory

Specifies that globals are stored in lower memory. The command implies large data model.

Syntax

```
-globalsInLowerMemory
-noglobalsInLowerMemory
```

3.13.13 -hprog | -hugeprog

Program memory compatibility is 21 bit.

Syntax

```
-[no]hprog | -[no]hugeprog
```

3.13.14 -initializedzerodata

Initializes zero globals in data instead of BSS.

Syntax

```
-initializedzerodata
-noinitializedzerodata
```

3.13.15 -ldata | -largedata

Specifies data space not limited to 64K.

Syntax

```
-[no]ldata | [no]largedata
```

3.13.16 -largeAddrInSdm

Generates index by 24-Bit Displacement (instead of 16-Bit) address register-indirect addressing mode, even in small data model.

This is required for accessing data memory above 0x007FFF with small data model.

Syntax

```
-[no]largeAddrInSdm
```

3.13.17 -min_enum_size

Specifies the minimum size for enumeration types.

Syntax

```
-min_enum_size keyword
```

The arguments of `keyword` are:

`1`

Minimum size is 1.

`2`

Minimum size is 2.

`4`

Minimum size is 4.

3.13.18 -padpipe

Controls pad pipeline for debugger.

Syntax

```
-padpipe
```

```
-nopadpipe
```

3.13.19 -profile

Generates code for profiling.

Syntax

```
-profile
-noprofile
```

3.13.20 -scheduling

Applies instruction scheduling.

Syntax

```
-scheduling
-noscheduling
```

3.13.21 -segchardata

Segregates character data.

Syntax

```
-segchardata
-nosegchardata
```

3.13.22 -sprog | -smallprog

Specifies program space limited to 64K.

Syntax

```
-[no]sprog | -[no]smallprog
```

3.13.23 -stackseq

Applies stack sequence optimization.

Syntax

```
-stackseq
-nostackseq
```

3.13.24 -strings

Controls how string literals are stored and used.

Remarks

```
-str[ings] keyword[, ...]
```

The keyword arguments are:

`[no]pool`

All string constants are stored as a single data object so your program needs one data section for all of them.

`[no]reuse`

All equivalent string constants are stored as a single data object so your program can reuse them. This is the default.

`[no]readonly`

Make all string constants read-only. This is the default.

3.13.25 -swp

Applies software pipelining.

Syntax

```
-swp
-noswp
```

3.13.26 -V3

Generates object file for 56800EX digital signal controller.

Syntax

```
-V3
-noV3
```

3.14 Optimization Options

The Optimization options are:

- -factor1
- -factor2
- -factor3
- -inline
- -ipa
- -nofactor1
- -nofactor2
- -nofactor3
- -O
- -O+
- -opt

3.14.1 -factor1

Turns on factorization step 1.

Syntax

```
-factor1
```

Remarks

To turn off factorization step 1, see -nofactor1.

3.14.2 -factor2

Turns on factorization step 2.

Syntax

```
-factor2
```

Remarks

To turn off factorization step 2, see -nofactor2.

3.14.3 -factor3

Turns on factorization step 3.

Syntax

```
-factor3
```

Remarks

To turn off factorization step 3, see -nofactor3.

3.14.4 -inline

Specify inline options. Default settings are `smart`, `noauto`.

Syntax

```
-inline keyword
```

The options for `keyword` are:

`none | off`

Turn off inlining.

`on | smart`

Turn on inlining for `inline` functions. This is the default.

`auto`

If `inline` not explicitly specified, auto-inline small functions.

`noauto`

Do not auto-inline. This is the default auto-inline setting.

`deferred`

Defer inlining until end of compilation unit. This allows inlining of functions in both directions.

`level=`*n*

Inline functions up to *n* levels deep. Level 0 is the same as `-inline on`. For `n`, enter 1 to 8 levels. This argument is case-sensitive.

`all`

Turn on aggressive inlining. This option is the same as `-inline on`, `-inline auto`.

`[no]bottomup`

Inline bottom-up starting from nodes of the call graph rather than the top-level function. This is the default.

3.14.5 -ipa

Specify Interprocedural Analysis Support (IPA) options.

Syntax

```
-ipa keyword[,...]
```

Select the interprocedural analysis level.

The `keyword` arguments are:

`function | off`

traditional mode (per function optimization)

`file`

per file optimization (same as `-deferred codegen`)

`program`

per program optimization. Use normally if compiling; pass all files or *.irobjs on the command line if linking.

`program-final|program2`

per program optimization. If not linking; pass all files or *.irobjs on the command line.

3.14.6 -nofactor1

Turns off factorization step 1.

Syntax

```
-nofactor1
```

Remarks

To turn on factorization step 1, see -factor1.

3.14.7 -nofactor2

Turns off factorization step 2.

Syntax

```
-nofactor2
```

Remarks

To turn on factorization step 2, see -factor2.

3.14.8 -nofactor3

Turns off factorization step 3.

Syntax

```
-nofactor3
```

Remarks

To turn on factorization step 3, see -factor3.

3.14.9 -O

Sets optimization settings to `-opt level=2`.

Syntax

```
-O
```

Remarks

Provided for backwards compatibility.

3.14.10 -O+

Controls optimization settings.

Syntax

```
-O+keyword [,...]
```

The `keyword` arguments are:

`0`

Equivalent to `-opt off`.

`1`

Equivalent to `-opt level=1`.

`2`

Equivalent to `-opt level=2`.

`3`

Equivalent to `-opt level=3`.

`4`

Equivalent to `-opt level=4, intrinsics`.

`p`

Equivalent to `-opt speed`.

`s`

Equivalent to `-opt space`.

Remarks

Options can be combined into a single command. Command is case-sensitive.

3.14.11 -opt

Specify code optimization options to apply to object code.

Remarks

```
-opt keyword [,...]
```

The `keyword` arguments are:

```
off | none
```

Suppress all optimizations. This is the default.

```
on
```

Same as `-opt level=2`

```
all | full
```

Same as `-opt speed, level=4, intrinsics, noframe`

```
l[evel]=num
```

Set a specific optimization level. The options for `num` are:

- `0` - Global register allocation only for temporary values. Prefix file equivalent: `#pragma optimization_level 0`.
- `1` - Adds dead code elimination, branch and arithmetic optimizations, expression simplification, and peephole optimization. Prefix file equivalent: `#pragma optimization_level 1`.
- `2` - Adds common subexpression elimination, copy and expression propagation, stack frame compression, stack alignment, and fast floating-point to integer conversions. Prefix file equivalent: `#pragma optimization_level 2`.
- `3` - Adds dead store elimination, live range splitting, loop-invariant code motion, strength reduction, loop transformations, loop unrolling (with -opt speed only), loop vectorization, lifetime-based register allocation, and instruction scheduling. Prefix file pragma equivalent: `optimization_level 3`.
- `4` - Like level 3, but with more comprehensive optimizations from levels 1 and 2. Prefix file equivalent: `#pragma optimization_level 4`.

For `num` options 0 through 4 inclusive, the default is 0.

`[no]space`

Optimize object code for size. Prefix file equivalent: `#pragma optimize_for_size on`.

`[no]speed`

Optimize object code for speed. Prefix file equivalent: `#pragma optimize_for_size off`.

`[no]cse | [no]commonsubs`

Common subexpression elimination. You can also add `#pragma opt_common_subs` to a prefix file.

`[no]deadcode`

Removal of dead code. Prefix file equivalent: `#pragma opt_dead_code`.

`[no]deadstore`

Removes dead assignments. Prefix file equivalent: `#pragma opt_dead_assignments`

`[no]lifetimes`

Computes variable lifetimes. Prefix file equivalent: `#pragma opt_lifetimes`

`[no]loop[invariants]`

Removes loop invariants. Prefix file equivalent: `#pragma opt_loop_invariants`

`[no]prop[agation]`

Propagation of constant and copy assignments. Prefix file equivalent: `#pragma opt_propagation`.

`[no]strength`

Strength reduction. Reducing multiplication by an array index variable to addition. Prefix file equivalent: `#pragma opt_strength_reduction`.

`[no]dead`

Same as `-opt [no]deadcode` and `[no]deadstore`. Prefix file equivalent: `#pragma opt_dead_code on|off` and `#pragma opt_dead_assignments`

`[no]peep[hole]`

Peephole optimization. Prefix file equivalent: `#pragma peephole`.

`[no]color[ing]`

Register coloring. Prefix file equivalent: `#pragma register_coloring`.

`[no]intrinsics`

Inlining of intrinsic functions.

`[no]schedule`

Perform instruction scheduling.

```
display | dump
```

Display complete list of active optimizations.

3.15 Debugging Control Options

The debugging control options are:

- -g
- -sym

3.15.1 -g

Generates debugging information.

Syntax

```
-g
```

Remarks

This command is global and case-sensitive. Same as `-sym full`. See -sym.

3.15.2 -sym

Specifies debugging options.

Syntax

```
-sym keyword[,...]
```

The arguments of `keyword` are:

`off`

Does not generate debugging information. This is the default.

`on`

Turns on debugging information.

```
full[path]
```

Stores full paths to source files.

Remarks

This command is global.

3.16 Assembler Control Options

The assembler control options are:

- -assert_nop
- -case
- -data
- -debug
- -debug_workaround
- -legacy
- -list
- -macro_expand
- -prog
- -warn_nop
- -warn_stall
- -warn_odd_sp
- -V3

3.16.1 -assert_nop

Adds NOP to resolve pipeline dependency. This is the default option.

Syntax

```
-[no]assert_nop
```

3.16.2 -case

Makes identifiers case-sensitive. This is the default option.

Syntax

```
-case
-nocase
```

3.16.3 -data

Provides data memory compatibility.

Syntax

```
-data keyword
```

The arguments of `keyword` are:

`16`

Represents 16-bit. This is the default one.

`24`

Represents 24-bit.

3.16.4 -debug

Generates debug information.

Syntax

```
-debug
-nodebug
```

3.16.5 -debug_workaround

Pads NOP workaround debugging issue in an implementation. This is the default option.

Syntax

```
-[no]debug_workaround
```

3.16.6 -legacy

Allows legacy DSP56800 instructions (data/prog 16).

Syntax

```
-[no]legacy
```

3.16.7 -list

Creates a listing file.

Syntax

```
-list
```

3.16.8 -macro_expand

Expands macro in listing output.

Syntax

```
-[no]macro_expand
```

3.16.9 -prog

Provides program memory compatibility.

Syntax

```
-prog keyword
```

The arguments of `keyword` are:

`16`

Represents 16-bit. This is the default one.

`19`

Represents 19-bit.

```
21
```

Represents 21-bit.

3.16.10 -warn_nop

Emits warning when there is a pipeline dependency.

Syntax

```
-[no]warn_nop
```

3.16.11 -warn_stall

Emits warning when there is a hardware stall.

Syntax

```
-[no]warn_stall
```

3.16.12 -warn_odd_sp

Issues warn instructions that increments/decrements odd amount to SP.

Syntax

```
-[no]warn_odd_sp
```

3.16.13 -V3

Supports 56800EX instructions.

Syntax

```
-[no]V3
```

3.17 Command Line Tools

This chapter includes the following sections:

3.17.1 Usage

To call the command-line tools, use the following format:

Table 3-1. Format

Tools	File Names	Format
Compiler	mwcc56800e.exe	compiler-options [linker-options] file-list
Linker	mwld56800e.exe	linker-options file-list
Assembler	mwasm56800e.exe	assembler-options file-list

The compiler automatically calls the linker by default and any options from the linker is passed on by the compiler to the assembler. However, you may choose to only compile with the -c flag. In this case, the assembler will only assemble and will not call the linker.

Also, available are environment variables. These are used to provide path information for includes or libraries, and to specify which libraries are to be included. You can specify the variables listed in the following table.

Table 3-2. Environment Variables

Tool	Library	Description
Compiler	MWC56800EIncludes	Similar to Access Paths panel; separate paths with ';' and prefix a path with '+' to specify a recursive path
Linker	MW56800ELibraries	Similar to MWC56800EIncludes
	MW56800ELibraryFiles	List of library names to link with project; separate with ';'
Assembler	MWAsm56800EIncludes	Similar to MWC56800EIncludes

These are the target-specific variables, and will only work with the DSP56800E tools. The generic variables **MWCIncludes**, **MWLibraries**, **MWLibraryFiles**, and **MWAsmIncludes** apply to all target tools on your system (such as Windows). If you only have the DSP56800E tools installed, then you may use the generic variables if you prefer.

3.17.2 Response File

In addition to specifying commands in the argument list, you may also specify a "response file". A response file's filename begins with an '@' (for example, @file), and the contents of the response file are commands to be inserted into the argument list. The response file supports standard UNIX-style comments. For example, the response file @file, contains the following:

Listing: Response file

```
# Response file @file

-o out.elf        # change output file name to 'out.elf'

-g                # generate debugging symbols
```

The above response file can used in a command such as:

mwcc56800e @file main.c

It would be the same as using the following command:

mwcc56800e -o out.elf -g main.c

3.17.3 Sample Build Script

The following is a sample of a DOS batch (BAT) file. The sample demonstrates:

- Setting of the environmental variables.
- Using the compiler to compile and link a set of files.

Listing: Sample DOS batch file

```
REM *** set CodeWarrior path ***

set CWFolder=C:\Freescale\CW MCU v10.x



REM *** set includes path ***
```

```
      set MWCIncludes=%CWFolder%\MCU\M56800E Support

      set MWLibraries=%CWFolder%\MCU\M56800E Support

      set MWLibraryFiles="%MWLibraries%\runtime_56800E\lib\Runtime 56800E.Lib" "%MWLibraries%
\msl\MSL_C\DSP_56800E\lib\MSL C 56800E.lib"



      REM *** add CLT directory to PATH ***

      set PATH=%PATH%;%CWFolder%\MCU\DSP56800x_EABI_Tools\Command_Line_Tools\



      REM *** compile options and files ***

      set COPTIONS=-c -I- -ir "%MWCIncludes%"
      set CFILELIST=file1.c file2.c
      set AOPTIONS=-c
      set AFILELIST=file3.asm file4.asm
      set LOPTIONS=-g -o output.elf
      set LFILELIST=file1.o file2.o file3.o file4.o
      set LCF=linker.cmd



      REM *** compile, assemble and link ***

      mwcc56800e %COPTIONS% %CFILELIST%

      mwasm56800e %AOPTIONS% %AFILELIST%

      mwld56800e %LOPTIONS% %LFILELIST% %LCF% %MWLibraryFiles%
```

3.17.4 Arguments

Listing: General command-Line options

```
---------------------------------------------------------------------------

General Command-Line Options
```

```
   All the options are passed to the linker   unless otherwise noted.

   Please see '-help usage' for details about the meaning of this help.

-------------------------------------------------------------------------

  -help [keyword[,...]]   # global; for this tool;
                          #   display help
     usage                #    show usage information
     [no]spaces           #    insert blank lines between options in
                          #      printout
     all                  #    show all standard options
     [no]normal           #    show only standard options
     [no]obsolete         #    show obsolete options
     [no]ignored          #    show ignored options
     [no]deprecated       #    show deprecated options
     [no]meaningless      #    show options meaningless for this target
     [no]compatible       #    show compatibility options
    opt[ion]=name         #    show help for a given option; for 'name',
                          #      maximum length 63 chars
    search=keyword        #    show help for an option whose name or help
                          #      contains 'keyword' (case-sensitive); for
                          #      'keyword', maximum length 63 chars
   group=keyword          #    show help for groups whose names contain
                          #     'keyword' (case-sensitive); for 'keyword'
                          #       maximum length 63 chars
   tool=keyword[,...]     #    categorize groups of options by tool;
```

```
                        #      default
   all                  #     show all options available in this tool
  this                  #     show options executed by this tool
                        #        default
    other|skipped       #      show options passed to another tool
    both                #      show options used in all tools
                        #
                        #
-version                #   global; for this tool;
                        #   show version, configuration, and build date
-timing                 #   global; collect timing statistics
-progress               #   global; show progress and version
-v[erbose]              #   global; verbose information; cumulative;
                        #   implies -progress
-search                 #  global; search access paths for source files
                        #   specified on the command line; may specify
                        #   object code and libraries as well; this
                        #   option provides the IDE's 'access paths'
                        #   functionality
-[no]wraplines          #   global; word wrap messages; default
-maxerrors max          # specify maximum number of errors to print, zero
                        #   means no maximum; default is 0
-maxwarnings max        #  specify maximum number of warnings to print,
```

```
                          #   zero means no maximum; default is 0

  -msgstyle keyword       #   global; set error/warning message style

     mpw                  #    use MPW message style

     std                  #    use standard message style; default

     gcc                  #    use GCC-like message style

     IDE                  #    use CW IDE-like message style

     parseable            #    use context-free machine-parseable message

                          #    style

                          #

  -[no]stderr             #  global; use separate stderr and stdout streams; if using -
nostderr, stderr goes to stdout



  -fullLicenseSearch      #  global; use more robust search for valid license files, will
result in somewhat longer build times
```

Listing: Compiler Options

```
---------------------------------------------------------------

Preprocessing, Precompiling, and Input File Control Options

---------------------------------------------------------------

  -c                  # global; compile only, do not link

  -[no]codegen        # global; generate object code

  -[no]convertpaths   # global; interpret #include filepaths specified

                      # for a foreign operating system; i.e.,

                      # <sys/stat.h> or <:sys:stat.h>; when enabled,

                      # '/' and ':' will separate directories and
```

```
                    # cannot be used in filenames (note:this is not a problem
                    # on Win32, since these characters are already disallowed
                    # in filenames; it is safe to leave the option 'on'); default
-cwd keyword         # specify #include searching semantics:  before
                    #   searching any access paths, the path
                    #   specified by this option will be searched
   proj             #    begin search in current working directory;
                    #      default
   source           #    begin search in directory of source file
   explicit         #    no implicit directory; only search '-I' or
                    #      '-ir' paths
   include          #    begin search in directory of referencing
                    #      file
                   #
-D+ | -d[efine      # cased; define symbol 'name' to 'value' if
  name[=value]      #   specified, else '1'
-[no]defaults       # global; passed to linker;
                    #   same as '-[no]stdinc'; default
-dis[assemble]      # global; passed to all tools;
                    #   disassemble files to stdout
-E                  # global; cased; preprocess source files
-EP                 # global; cased; preprocess and strip out #line/#pragma directives


-enc[oding] keyword          # specify default source encoding; compiler  automatically
```

```
detects UTF-8 header or UCS-2/UCS-4 encodings regardless of setting



    ascii                       # ASCII; default

    autodetect|multibyte|mb     # scan file for multibyte_encoding (slower)

                                #

    system                      # use system locale

    UTF[8|-8]                   # UTF-8

    SJIS|Shift-JIS|ShiftJIS     # Shift-JIS

    EUC[JP|-JP]                 # EUC-JP

    ISO[2022JP|-2022-JP]        # ISO-2022-JP



  -ext extension                # global; specify extension for  generated object files; with
a leading period ('.'), appends  extension; without, replaces source file's extension; for
'extension', maximum length 14 chars; default is none



  -gccinc[ludes]                # global; adopt GCC #include semantics: add '-I' paths to
system list if '-I-' is not specified, and search  directory of referencing file first for
#includes (same as -cwd include)



-[no]gccdep[ends]               # global; if set, write dependency file (-MD, -MMD) with name
and location based on output file (compatible with gcc 3.x); else base filename on the
source file and write to the current directory (legacy MW behavior)



  -i- | -I-                     # global; change target for '-I' access paths to the system
list; implies '-cwd explicit'; while compiling, user paths then system paths are searched
when using '#include "..."; only system paths are searched with '#include <...>'



  -I+ | -i path                 # global; cased; append access path to current #include list
(see '-gccincludes' and '-I-')
```

```
  -include file               # prefix text file or precompiled header onto all source files


  -ir path                    # global; append a recursive access path to current #include
list


  -[no]keepobj[ects]          # global; keep object files generated after invoking linker;
if disabled, intermediate object files are temporary and deleted after link stage; objects
are always kept when compiling




  -M                # global; cased; scan source files for

                    # dependencies and emit Makefile, do not

                    # generate object code

  -MM               # global; cased; like -M, but do not list system

                    # include files

  -MD               # global; cased; like -M, but write dependency

                    # map to a file (see ~gccdep) and generate object code



  -MMD              # global; cased; like -MD, but do not list system

                    # include files

  -Mfile file       # global; cased; like -M, but write dependency map to the specified
file

  -MMfile file      # global; cased; like -MM, but write dependency

                    # map to the specified file

  -MDfile file      # global; cased; like -MD, but write dependency

                    # map to the specified file
```

```
  -MMDfile file      # global; cased; like -MMD, but write dependency

                     # map to the specified file

  -make              # global; scan source files for dependencies
and                                  # emit Makefile, do not generate object code

  nofail             # continue working after errors in earlier files

  -nolink            # global; compile only, do not link

  -noprecompile      # do not precompile any files based on the

                     # filename extension

  -nosyspath         # global; treat #include <...> like #include

                     # "..."; always search both user and system

                     # path lists

  -o file|dir        # specify output filename or directory for object

                     # file(s) or text output, or output filename

                     # for linker if called

  -P                 # global; cased; preprocess and send output to

                     # file; do not generate code

 -precompile file|dir# generate precompiled header from source; write

                     # header to 'file' if specified, or put header

                     # in 'dir'; if argument is "", write header to

                     # source-specified location; if neither is

                     # defined, header filename is derived from

                     # source filename; note: the driver can tell

                     # whether to precompile a file based on its

                     # extension; '-precompile file source' then is
```

```
                    # the same as '-c -o file source'
-preprocess         # global; preprocess source files


-ppopt keyword[,...]# specify options affecting the preprocessed output


  [no]break         # emit file/line breaks; default
  [no]line          # emit #line directives, else comments; default
  [no]full[path]    # emit full path of file, else base filename; default


  [no]pragma        # keep #pragma directives, else strip them; default


  [no]comment       # keep comments, else strip them
  [no]space         # keep whitespace, else strip it; default


-prefix file        # prefix text file or precompiled header onto all
                    # source files
-S                  # global; cased; passed to all tools;
                    #   disassemble and send output to file
-[no]stdinc         # global; use standard system include paths
                    # (specified by the environment variable
                    # %MWCIncludes%); added after all system '-I'
                    # paths; default
-U+ | -u[ndefine] name    # cased; undefine symbol 'name'
```

```
  -allow_macro_redefs       # allow macro redefinitions without an error or warning



-------------------------------------------------------------------------

Front-End C/C++ Language Options

-------------------------------------------------------------------------

  -ansi keyword      # specify ANSI conformance options, overriding

                     #   the given settings

     off             #    same as '-stdkeywords off', '-enum min', and

                     #      '-strict off'; default

     on|relaxed      #    same as '-stdkeywords on', '-enum min', and

                     #      '-strict on'

     strict          #    same as '-stdkeywords on', '-enum int', and

                     #      '-strict on'

                     #

  -nolonglong        # disable 'long long' support

  -char keyword      # set sign of 'char'

     signed          #    chars are signed; default

     unsigned        #    chars are unsigned

  -enum keyword      # specify default size for enumeration types

     min             # use the minimal-sized type; default

     int             # use int-sized enums

                     #

  -min_enum_size keyword# specify the minimum size for enumeration types (implies -enum min)
```

```
    1               # minimum size is 1

    2               # minimum size is 2

    4               # minimum size is 4

                    #

  -for_scoping on|off   # control legacy (non-standard) for-scoping behavior; when enabled,
variables declared in 'for' loops are visible to the enclosing scope; when disabled, such
variables are scoped to the loop only; default is off




  -fl[ag] pragma        # specify an 'on/off' compiler #pragma; '-flag  foo' is the same as
'#pragma foo on', '-flag no-foo' is the same as '#pragma foo off'; use '-pragma' option for
other cases




  -inline keyword[,...] # specify inline options

    on|smart            #    turn on inlining for 'inline' functions;

                        #      default

    none|off            #    turn off inlining

    auto                #    auto-inline small functions (without

                        #      'inline' explicitly specified)

    noauto              #    do not auto-inline; default

    all                 #    turn on aggressive inlining: same
as                                          #      '-inline on, auto'

    deferred            #    defer inlining until end of compilation

                        #   unit; this allows inlining of functions

                        #   defined before and after the caller;

                        #   deprecated option, use '-ipa file'
```

```
    level=n             #    cased; inline functions up to 'n' levels

                        #   deep; level 0 is the same as '-inline on';

                        #      for 'n', range 0 - 8

    [no]bottomup        # inline bottom-up, starting from leaves of the call graph rather
than the top-level function; default



  -ipa keyword[,...]    # select interprocedural analysis level

    function|off        # traditional mode (per-functionoptimization); default;

    file                # per-file optimization (same as 'deferred codegen')

    program             # per-program optimization (if compiling, use  normally; if linking,
pass all files or *.irobjs on the command line)



    program-final|program2 # per-program optimization (without linking; pass all files or
*.irobjs on the command line)



  -[no]mapcr            # reverse mapping of '\n' and '\r' so that

                        #   '\n'==13 and '\r'==10 (for Macintosh MPW

                        #   compatability)

  -once                 # prevent header files from being processed more than once



  -pragma               # specify a #pragma for the compiler such as "#pragma ..."; quote
the parameter if you provide an argument (i.e., '-pragma "myopt reset"')



  -r[equireprotos]      # require prototypes

  -relax_pointers       # relax pointer type-checking rules in C
```

```
  -stdkeywords on|off   # allow only standard keywords; default is off

  -str[ings] keyword[,...]  # specify string constant options

     [no]reuse          # reuse strings; equivalent strings are the same object; default



     [no]pool           # pool strings into a single data object

     [no]readonly       # make all string constants read-only

                      #

  -strict on|off        # specify ANSI strictness checking; default is off



  -trigraphs on|off     # enable recognition of trigraphs; default is off



-------------------------------------------------------------------------

Optimizer Options



   Note that all options besides '-opt off|on|all|space|speed|level= ...' are for backwards
compatibility; other optimization options may be superceded by use of '-opt level=xxx'.

-------------------------------------------------------------------------

  -O                # same as '-O2'

  -O+keyword[,...]  # cased; control optimization; you may combine

                    #   options as in '-O4,p'

     0              #    same as '-opt off'

     1              #    same as '-opt level=1'

     2              #    same as '-opt level=2'

     3              #    same as '-opt level=3'
```

```
    4               #    same as '-opt level=4'
    p               #    same as '-opt speed'
    s               #    same as '-opt space'
                    #
-opt keyword[,...] # specify optimization options
    off|none        #    suppress all optimizations; default
    on              #    same as '-opt level=2'
    all|full        #    same as '-opt speed, level=4'
    [no]space       #    optimize for space
    [no]speed       #    optimize for speed
    l[evel]=num     #    set optimization level:
                    #    level 0: no optimizations
                    #
                    #    level 1: global register allocation,
                    #    peephole, dead code elimination
                    #
                    #    level 2: adds common subexpression
                    #    elimination and copy propagation
                    #
                    #    level 3: adds loop transformations,
                    #    strength reduction, loop-invariant code
                    #    motion
                    #
```

```
                    #    level 4: adds repeated common
                    #    subexpression elimination and
                    #    loop-invariant code motion
                    #    ; for 'num', range 0 - 4; default is 0
    [no]cse         #    common subexpression elimination
    [no]commonsubs  #
    [no]deadcode    #    removal of dead code
    [no]deadstore   #    removal of dead assignments
    [no]lifetimes   #    computation of variable lifetimes
    [no]loop[invariants]   #    removal of loop invariants
    [no]prop[agation]  #  propagation of constant and copy assignments
    [no]strength    #   strength reduction; reducing multiplication
                    #      by an index variable into addition
     [no]dead       #    same as '-opt [no]deadcode' and '-opt
                    #      [no]deadstore'
     display|dump   #    display complete list of active
                    #      optimizations
                    #


------------------------------------------------------------------------

DSP M56800E CodeGen Options

------------------------------------------------------------------------

  -DO keyword                 # for this tool;
                              # specify hardware DO loops
```

```
    off                         # no hardware DO loops; default

    nonested                    # hardware DO loops but no nested ones

    nested                      # nested hardware DO loops

                                #

  -[no]padpipe                  # for this tool;

                                # pad pipeline for debugger; default

  -[no]ldata | -[no]largedata   # passed to linker;

                                #  data space not limited to 64K

  -[no]globalsInLowerMemory     # for this tool;

                                # globals live in lower memory; implies

                                # '-large data model'

  -[no]largeAddrInSdm           # for this tool;
                                # Index by 24-Bit Displacement (Instead of
                                # 16-Bit) address Register-Indirect Addressing
                                # mode generated even in small data model.
                                # Required for accessing X data memory above
                                # 0x007FFF with small data model.

  -[no]sprog | -[no]smallprog   # for this tool;

                                #   program space limited to 64K

  -[no]hprog | -[no]hugeprog    # for this tool; program memory

                                # compatibility - 21 bit.

  -[no]segchardata              # for this tool;

                                #   segregate character data

  -[no]initializedzerodata      # for this tool;

                                # initialized zero globals in data instead of BSS

  -[no]asmout                  # for this tool;

                               #   assembly file output
```

```
-chkasm keyword             # for this tool;
                            #  check for pipeline in inline assembly
                            # sources
   off                      # no check; default
   conflict                 # error on pipeline conflict
   conflict_and_stall       # error on pipeline conflict, warning on
                            #    hardware stalls
-chkcsrcpipeline keyword    # for this tool;
                            #   check for pipeline in C sources
   off                      # no check; default
   conflict                 # error on pipeline conflict
                            #
-[no]factor1                # for this tool;
                            #   applies factorization 1
-[no]factor2                # for this tool;
                            #   applies factorization 2
-[no]factor3                # for this tool;
                            #   applies factorization 3
-[no]profile                # for this tool;
                            #   generate code for profiling
-[no]scheduling             # for this tool;
                            #   applies instruction scheduling
-[no]allowREP               # for this tool;
```

```
                              #   REP instruction generation when appropriate


  -[no]swp                    # for this tool;
                              #   applies software pipelining
  -[no]stackseq               # for this tool;
                              #   applies stack sequence optimization
  -[no]constarray             # for this tool;
                              #   applies constant to array optimization
  -[no]v3                     # for this tool;
                              #   generate object file for 56800EX digital signal controller


-------------------------------------------------------------------------

Debugging Control Options

-------------------------------------------------------------------------

  -g                      # global; cased; generate debugging information;
                          #   same as '-sym full'
  -sym keyword[,...]      # global; specify debugging options
     off                  #    do not generate debugging information;
                          #      default
     on                   #    turn on debugging information
     full[path]           #    store full paths to source files
                          #

-------------------------------------------------------------------------
```

```
C/C++ Warning Options

-------------------------------------------------------------------------

  -w[arn[ings]]           # global; for this tool;

    keyword[,...]         #   warning options

     off                  #    passed to all tools;

                          #      turn off all warnings

     on                   #    passed to all tools;

                          #      turn on most warnings

     [no]cmdline          #    passed to all tools;

                          #      command-line driver/parser warnings

     [no]err[or] |        #    passed to all tools;

       [no]iserr[or]      #      treat warnings as errors

     most                 #    turn on most warnings

     all                  #    turn on almost all warnings, require

                          #      prototypes

     full                 #    turn on all warnings (likely to generate

                          #      spurious warnings), require prototypes

     [no]pragmas |        #    illegal #pragmas (most)

       [no]illpragmas     #

     [no]empty[decl]      #    empty declarations (most)

     [no]possible |       #    possible unwanted effects (most)

       [no]unwanted       #

     [no]unusedarg        #    unused arguments (most)

     [no]unusedvar        #    unused variables (most)
```

```
[no]unused          #    same as -w [no]unusedarg,[no]unusedvar
                    #      (most)
[no]extracomma |    #    extra commas (most)
  [no]comma         #
[no]pedantic |      #    pedantic error checking (most)
  [no]extended      #
[no]hidevirtual |   #    hidden virtual functions (most)
  [no]hidden[virtual]
[no]largeargs       #    passing large arguments to unprototyped
                    #      functions (most)
[no]unusedexpr      #    use of expressions as statements without
                    #      side effects (most)
[no]ptrintconv      #   lossy conversions from pointers to
                    #      integers (most)
[no]tokenpasting    #   token not formed by ## operator (most)
[no]missingreturn   #   return without a value in non-void-
                    #      returning function  (most)
[no]structclass     #    inconsistent use of 'class' and 'struct'
                    #      (most)
[no]filecaps        # incorrect capitalization used in #include
                    #   "..." (most)
[no]sysfilecaps     # incorrect capitalization used in #include
                    #    <...> (most)
```

```
      [no]implicit[conv]   # implicit arithmetic conversions; implies
                           # '-warn impl_float2int,impl_signedunsigned'
                           #   (all)
      [no]impl_int2float   # implicit integral to floating conversions
                           #   (all)
      [no]impl_float2float # implicit float to floating conversions
                           #   (all)
      [no]impl_signedunsigned  # implicit signed/unsigned conversions (all)
      [no]relax_i2i_conv        # relax warnings for implicit integer to
                           #   integer arithmetic conversions (off for
                           #   full, on otherwise)
[no]undef[macro]           # use of undefined macros in #if/#elif
                           #   conditionals (full)
      [no]notinlined       # 'inline' functions not inlined (full)
      [no]padding          # padding added between struct members(full)
      [no]notused          # result of non-void-returning function not
                           #   used (full)
      [no]anyptrintconv    # any conversions from pointers to integers (full)

      [no]alias_ptr_conv   # generates warnings for potentially dangerous pointer casts (full)

      display|dump         # display list of active warnings
                          #
```

Listing: Command-line Linker options

```
-------------------------------------------------------------------

Command-Line Linker Options

-------------------------------------------------------------------


  -dis[assemble]     # global; disassemble object code and do not
                     #   link; implies '-nostdlib'
  -L+ | -l path      # global; cased; add library search path; default
                     #   is to search current working directory and
                     #   then system directories (see '-defaults');
                     #   search paths have global scope over the
                     #   command line and are searched in the order
                     #   given
  -lr path           # global; like '-l', but add recursive library
                     #   search path
  -l+file            # cased; add a library by searching access paths
                     #   for file named lib<file>.<ext> where <ext> is
                     #   a typical library extension; if that fails,
                     #   try to add <file> directly; library added in
                     #   link order before system libraries (see
                     #   '-defaults')
  -[no]defaults      # global; same as -[no]stdlib; default
  -nofail            # continue importing or disassembling after
                     #   errors in earlier files
```

```
  -[no]stdlib         # global; use system library access paths
                      #   (specified by %MWLibraries%) and add system
                      #   libraries (specified by %MWLibraryFiles%) at
                      #   end of link order; default
  -reverselibsearchpath  # global; reverse search order of library paths


  -S                  # global; cased; disassemble and send output to
                      #   file; do not link; implies '-nostdlib'


-------------------------------------------------------------------------
ELF Linker Options
-------------------------------------------------------------------------
  -[no]dead[strip]    # enable dead-stripping of unused code; default
  -force_active       # specify a list of symbols as undefined; useful
    symbol[,...]      #   to force linking of static libraries
                      #
  -keep[local] on|off  # keep local symbols (such as relocations and
                      #   output segment names) generated during link;
                      #   default is on
  -m[ain] symbol      # set main entry point for application or shared
                      #   library; use '-main ""' to specify no entry
                      #   point; for 'symbol', maximum length 63 chars;
                      #   default is 'FSTART_'
```

```
-map [keyword[,...]]   # generate link map file
   closure          #    calculate symbol closures
   unused           #    list unused symbols
   showbyte         #    show byte relocation used on symbols
                    #
-sortbyaddr         # sort S-records by address; implies '-srec'
-srec               # generate an S-record file; ignored when
                    #   generating static libraries
-sreceol keyword    # set end-of-line separator for S-record file;
                    #   implies '-srec'
   mac              #    Macintosh ('\r')
   dos              #    DOS ('\r\n'); default
   unix             #    Unix ('\n')
                    #
-sreclength length  # specify length of S-records (should be a
                    #   multiple of 4); implies '-srec'; for
                    #   'length', range 8 - 252; default is 64
-usebyteaddr        # use byte address in S-record file; implies
                    #   '-srec'
-V3                 # generate elf file for 56800EX digital signal
                    #   controller
-o file             # specify output filename
```

```
-------------------------------------------------------------------------

DSP M56800E Project Options

-------------------------------------------------------------------------

  -application        # global; generate an application; default

  -library            # global; generate a static library



-------------------------------------------------------------------------

DSP M56800E CodeGen Options

-------------------------------------------------------------------------

  -[no]ldata |        # data space not limited to 64K

    -[no]largedata    #



-------------------------------------------------------------------------

Linker C/C++ Support Options

-------------------------------------------------------------------------

  -Cpp_exceptions on|off    # enable or disable C++ exceptions; default is on



  -dialect | -lang keyword  # specify source language

     c                      #  treat source as C++ unless its extension is '.c', '.h', or
'.pch'; default



     c++                    #  treat source as C++ always

                            #
```

```
-------------------------------------------------------------------

Debugging Control Options

-------------------------------------------------------------------

  -g                  # global; cased; generate debugging information;
                      #   same as '-sym full'
  -sym keyword[,...]  # global; specify debugging options
     off              #   do not generate debugging information;
                      #     default
     on               #   turn on debugging information
     full[path]       #   store full paths to source files
                      #

-------------------------------------------------------------------

Warning Options

-------------------------------------------------------------------

  -w[arn[ings]]             # global; warning options
    keyword[,...]           #
     off                    #    turn off all warnings
     on                     #    turn on all warnings
     [no]cmdline            #    command-line parser warnings
     [no]err[or] |          #    treat warnings as errors
       [no]iserr[or]        #
     noSymRedef             #    suppress Symbol Redefined warnings
     display|dump           #    display list of active warnings
                            #
```

```
-------------------------------------------------------------------------

ELF Disassembler Options

-------------------------------------------------------------------------

  -show keyword[,...]     # specify disassembly options

     only|none            #    as in '-show none' or, e.g.,

                          #      '-show only,code,data'

     all                  #    show everything; default

   [no]code | [no]text    #  show disassembly of code sections; default

     [no]comments         #  show comment field in code; implies '-show

                          #      code'; default

     [no]extended         #    show extended mnemonics; implies '-show

                          #      code'; default

     [no]data             #    show data; with '-show verbose', show hex

                          #      dumps of sections; default

   [no]debug | [no]sym    #    show symbolics information; default

   [no]exceptions         #  show exception tables; implies '-show data';

                          #      default

     [no]headers          #    show ELF headers; default

     [no]hex              #    show addresses and opcodes in code

                          #    disassembly; implies '-show code'; default

     [no]names            #    show symbol table; default

     [no]relocs           #    show resolved relocations in code and
```

```
                        #      relocation tables; default

     [no]source         #    show source in disassembly; implies '-show

                        #    code'; with '-show verbose', displays

                        #      entire source file in output, else shows

                        #      only four lines around each function;

                        #      default

     [no]xtables        #    show exception tables; default

     [no]verbose        #  show verbose information, including hex dump

                        #    of program segments in applications;

                        #    default

  -dispaths =           #    disassembler file paths mapping, useful to map libraries
sources, src=dest
```

Listing: Assembler Control Options

```
-------------------------------------------------------------------

Assembler Control Options

-------------------------------------------------------------------

  -[no]case           # identifiers are case-sensitive; default

  -[no]debug          # generate debug information

  -list               # create a listing file

  -[no]macro_expand   # expand macro in listin output

  -[no]assert_nop     # add nop to resolve pipeline dependency; default

  -[no]warn_nop       # emit warning when there is a pipeline

                      #   dependency

  -[no]warn_stall     # emit warning when there is a hardware stall
```

```
-[no]legacy          # allow legacy DSP56800 instructions(imply
                     #   data/prog 16)
-[no]debug_workaround  # Pad nop workaround debuggin issue in some
                     #   implementation; default
-data keyword        # data memory compatibility
   16                #    16 bit; default
   24                #    24 bit
                     #
-prog keyword        # program memory compatibility
   16                #    16 bit; default
   19                #    19 bit
   21                #    21 bit
-[no]warn_odd_sp     # Warn instructions that increment/decrement odd
                     #    amount to SP
-[no]V3              # Support 56800EX instructions
                     #
```

Chapter 4
C for DSP56800E

This chapter explains considerations for using C with the DSP56800E processor.

This chapter includes the following sections:

4.1 Data Types

This section explains how the CodeWarrior compiler implements ordinal and floating-point number types for 56800E processors. For more information, read `limits.h` and `float.h`, in the M56800E Support folder.

4.1.1 Ordinal Data Types

The following table shows the sizes and ranges of ordinal data types.

Table 4-1. 56800E Ordinal Types

Type	Option Settings	Size (bits)	Range
`char`	**Use Unsigned Char**s is disabled in the C/C++ Language (C Only) settings panel	8	`-128 to 127`
	Use Unsigned Chars is enabled	8	`0 to 255`
`signed char`	n/a	8	`-128 to 127`
`unsigned char`	n/a	8	`0 to 255`
`short`	n/a	16	`-32,768 to 32,767`
`unsigned short`	n/a	16	`0 to 65,535`
`int`	n/a	16	`-32,768 to 32,767`
`unsigned int`	n/a	16	`0 to 65,535`
`long`	n/a	32	`-2,147,483,648 to 2,147,483,647`
`unsigned long`	n/a	32	`0 to 4,294,967,295`
`pointer`	small data model (“Large Data Model” is disabled in the M56800E Processor settings panel)	16	`0 to 65,535`
	large data model (“Large Data Model” is enabled)	24	`0 to 16,777,215`

4.1.2 Floating Point Types

The following table shows the sizes and ranges of the floating-point types.

Table 4-2. M56800E Floating-Point Types

Type	Size (bits)	Range
`float`	32	`1.17549e-38 to 3.40282e+38`
`short double`	32	`1.17549e-38 to 3.40282e+38`
`double`	32	`1.17549e-38 to 3.40282e+38`
`long double`	32	`1.17549e-38 to 3.40282e+38`

4.1.3 64-Bit Data Types

The compiler supports 64-bit `long` and double data types, although 32-bit `long` and `double` data types are the default. To activate and use the 64-bit data types, you must:

1. Use #pragma slld on in a common header file, or in the C/C++ Preprocessor panel.

2. Use precompiled Main Standard Library (MSL) and runtime support libraries with the _SLLD suffix (for example, use MSL C 56800E smm_SLLD.lib instead of MSL C 56800E smm.lib and runtime 56800E smm_SLLD.lib instead of runtime 56800E smm.lib).
3. Add * (ll_engine.text) to the code section in the linker command file.

4.2 Calling Conventions and Stack Frames

The DSP56800E compiler stores data and call functions differently than the DSP56800 compiler does. Advantages of the DSP56800E method include: more registers for parameters and more efficient byte storage.

This topic contains the following sub-topics:

- Passing Values to Functions
- Returning Values From Functions
- Volatile and Non-Volatile Registers
- Stack Frame and Alignment

4.2.1 Passing Values to Functions

The compiler uses registers A,B, R1, R2, R3, R4, Y0, and Y1 to pass parameter values to functions. Upon a function call, the compiler scans the parameter list from left to right, using registers for these values:

- The first two 8/16-bit integer values — Y0 and Y1.
- The first two 32-bit integer or float values — A and B.
- The first four pointer parameter values — R2, R3, R4, and R1 (in that order).
- The third and fourth 8/16-bit integer values — A and B (provided that the compiler does not use these registers for 32-bit parameter values).
- The third 8/16-bit integer value — B (provided that the compiler does not use this register for a 32-bit parameter value).

The compiler passes the remaining parameter values on the stack. The system increments the stack by the total amount of space required for memory parameters. This incrementing must be an even number of words, as the stack pointer (SP) must be continuously long-aligned. The system moves parameter values to the stack from left to right, beginning with the stack location closest to the SP. Because a long parameter must begin at an even address, the compiler introduces one-word gaps before long parameter values, as appropriate.

4.2.2 Returning Values From Functions

The compiler returns function results in registers A, R0, R2, and Y0:

- 8-bit integer values — Y0.
- 16-bit integer values — Y0.
- 32-bit integer or float values — A.
- All pointer values — R2.
- Structure results — R0 contains a pointer to a temporary space allocated by the caller. (The pointer is a hidden parameter value.)

Additionally, the compiler:

- Reserves R5 for the stack frame pointer when a function makes a dynamic allocation. (This is the original stack pointer before allocations.) Otherwise, the compiler saves R5 across function calls.
- Saves registers C10 and D10 across function calls.
- Does not save registers C2 and D2 across function calls.

4.2.3 Volatile and Non-Volatile Registers

Values in *non-volatile* registers can be saved across functions calls. Another term for such registers is *saved over a call* registers (SOCs).

Values in *volatile* registers cannot be saved across functions calls. Another term for such registers is *non-SOC* registers.

The following table lists both the volatile and non-volatile registers.

Table 4-3. Volatile and Non-Volatile Registers

Unit	Register	Size	Type	Comments
Arithmetic Logic Unit (ALU)	Y1	16	Volatile (non-SOC)	
	Y0	16	Volatile (non-SOC)	
	Y	32	Volatile (non-SOC)	
	X0	16	Volatile (non-SOC)	
	A2	4	Volatile (non-SOC)	
	A1	16	Volatile (non-SOC)	
	A0	16	Volatile (non-SOC)	
Arithmetic Logic Unit (ALU)	A10	32	Volatile (non-SOC)	

Table continues on the next page...

Table 4-3. Volatile and Non-Volatile Registers (continued)

Unit	Register	Size	Type	Comments
	A	36	Volatile (non-SOC)	
	B2	4	Volatile (non-SOC)	
	B1	16	Volatile (non-SOC)	
	B0	16	Volatile (non-SOC)	
	B10	32	Volatile (non-SOC)	
	B	36	Volatile (non-SOC)	
	C2	4	Volatile (non-SOC)	
	C1	16	Non-Volatile (SOC)	
	C0	16	Non-Volatile (SOC)	
	C10	32	Non-Volatile (SOC)	
	C	36	Volatile (non-SOC)	Includes volatile register C2.
	D2	4	Volatile (non-SOC)	
	D1	16	Non-Volatile (SOC)	
	D0	16	Non-Volatile (SOC)	
	D10	32	Non-Volatile (SOC)	
	D	36	Volatile (non-SOC)	Includes volatile register D2.
Address Generation Unit (AGU)	R0	24	Volatile (non-SOC)	
Address Generation Unit (AGU) *(continued)*	R1	24	Volatile (non-SOC)	
	R2	24	Volatile (non-SOC)	
	R3	24	Volatile (non-SOC)	
	R4	24	Volatile (non-SOC)	
	R5	24	Non-volatile (SOC)	If the compiler uses R5 as a pointer, it becomes a non-volatile register — its value can not be saved over called functions.
	N	24	Volatile (non-SOC)	
	SP	24	Volatile (non-SOC)	
	N3	16	Volatile (non-SOC)	
	M01	16	Volatile (non-SOC)	Certain registers must keep specific values for proper C execution — set this register to 0xFFFF.
Program Controller	PC	21	Volatile (non-SOC)	
	LA	24	Volatile (non-SOC)	
	LA2	24	Volatile (non-SOC)	
	HWS	24	Volatile (non-SOC)	

Table continues on the next page...

Table 4-3. Volatile and Non-Volatile Registers (continued)

Unit	Register	Size	Type	Comments
	FIRA	21	Volatile (non-SOC)	
	FISR	13	Volatile (non-SOC)	
Program Controller *(continued)*	OMR	16	Volatile (non-SOC)	Certain registers must keep specific values for proper C execution — in this register, set the CM bit.
	SR	16	Volatile (non-SOC)	
	LC	16	Volatile (non-SOC)	
	LC2	16	Volatile (non-SOC)	

4.2.4 Stack Frame and Alignment

The following figure depicts generation of the stack frame. The stack grows upward, so pushing data onto the stack increments the stack pointer's address value.

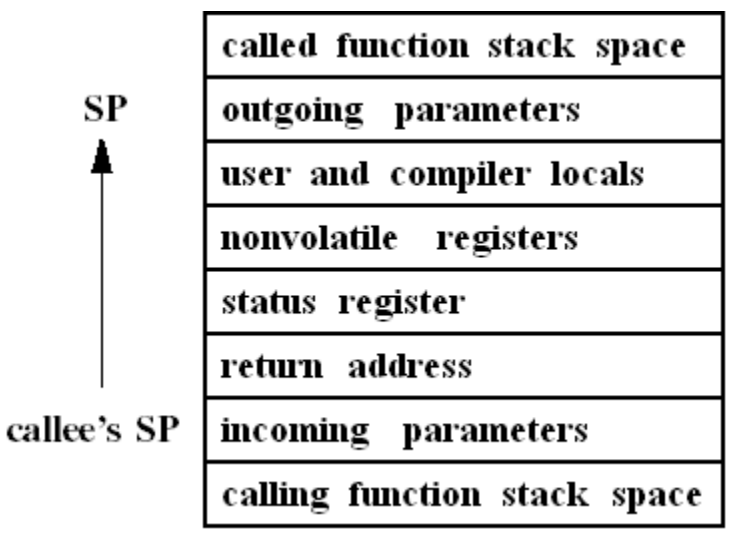

Figure 4-1. Stack Frame

The stack pointer (SP) is a 24-bit register, always treated as a word pointer. During a function execution, the stable position for the SP is at the top of the user and compiler locals. The SP increases during the call if the stack is used for passed parameters.

The software stack supports structured programming techniques, such as parameter passing to subroutines and local variables. These techniques are available for both assembly-language and high-level-language programming. It is possible to support passed parameters and local variables for a subroutine at the same time within the stack frame.

The compiler stores local data by size. It stores smaller data closest to the SP, exploiting SP addressing modes that have small offsets. This means that the compiler packs all bytes two per word near the stack pointer. It packs the block of words next, then blocks of longs. Aggregates (structs and arrays) are farthest from the stack pointer, not sorted by size.

NOTE

When a function makes a dynamic allocation, the compiler reserves R5 as a stack frame pointer. (This is the stack pointer before allocations.)

The compiler always must operate with the stack pointer long aligned. This means that:

- The start-up code in the runtime first initializes the stack pointer to an odd value.
- At all times after that, the stack pointer must point to an odd word address.
- The compiler never generates an instruction that adds or subtracts an odd value from the stack pointer.
- The compiler never generates a MOVE.W or MOVEU.W instruction that uses the X:(SP)+ or X:(SP)- addressing mode.

4.3 User Stack Allocation

The 56800E compilers build frames for hierarchies of function calls using the stack pointer register (SP) to locate the next available free X memory location in which to locate a function call's frame information. There is usually no explicit frame pointer register. Normally, the size of a frame is fixed at compile time. The total amount of stack space required for incoming arguments, local variables, function return information, register save locations (including those in pragma interrupt functions) is calculated and the stack frame is allocated at the beginning of a function call.

Sometimes, you may need to modify the SP at runtime to allocate temporary local storage using inline assembly calls. This invalidates all the stack frame offsets from the SP used to access local variables, arguments on the stack, etc. With the User Stack Allocation feature, you can use inline assembly instructions (with some restrictions) to modify the SP while maintaining accurate local variable, compiler temps, and argument offsets, that is these variables can still be accessed since the compiler knows you have modified the stack pointer.

The User Stack Allocation feature is enabled with the `#pragma check_inline_sp_effects [on|off|reset]` pragma setting. The pragma may be set on individual functions. By default the pragma is off at the beginning of compilation of each file in a project.

The User Stack Allocation feature allows you to simply add inline assembly modification of the SP anywhere in the function. The restrictions are straight-forward:

1. The SP must be modified by the same amount on all paths leading to a control flow merge point.
2. The SP must be modified by a literal constant amount. That is, address modes such as “(SP)+N” and direct writes to SP are not handled.
3. The SP must remain properly aligned.
4. You must not overwrite the compiler’s stack allocation by decreasing the SP into the compiler allocated stack space.

Point 1 above is required when you think about an if-then-else type statement. If one branch of a decision point modifies the SP one way and the other branch modifies SP another way, then the value of the SP is run-time dependent, and the compiler is unable to determine where stack-based variables are located at run-time. To prevent this from happening, the User Stack Allocation feature traverses the control flow graph, recording the inline assembly SP modifications through all program paths. It then checks all control flow merge points to make sure that the SP has been modified consistently in each branch converging on the merge point. If not, a warning is emitted citing the inconsistency.

Once the compiler determined that inline SP modifications are consistent in the control flow graph, the SP’s offsets used to reference local variables, function arguments, or temps are fixed up with knowledge of inline assembly modifications of the SP. You may freely allocate local stack storage:

1. As long as it is equally modified along all branches leading to a control flow merge point.
2. The SP is properly aligned. The SP must be modified by an amount the compiler can determine at compile time.

A single new pragma is defined. `#pragma check_inline_sp_effects [on|off|reset]` will generate a warning if the user specifies an inline assembly instruction which modifies the SP by a run-time dependent amount. If the pragma is not specified, then stack offsets used to access stack-based variables will be incorrect. It is the user’s responsibility to enable `#pragma check_inline_sp_effects`, if they desire to modify the SP with inline assembly and access local stack-based variables. This pragma has no effect in function level assembly functions or separate assembly only source files (`.asm` files).

In general, inline assembly may be used to create arbitrary flow graphs and not all can be detected by the compiler.

For example:

```
REP #3

ADDA #2,SP
```

This example would modify the SP by three, but the compiler would only see a modification of one. Other cases such as these might be created by the user using inline jumps or branches. These are dangerous constructs and are not detected by the compiler.

In cases where the SP is modified by a run-time dependent amount, a warning is issued.

Listing: Example 1 - Legal Modification of SP Using Inline Assembly

```
#define EnterCritical() { asm(adda #2,SP);\
                          asm(move.l  SR,X:(SP)+); \
                          asm(bfset  #0x0300,SR); \
                          asm(nop); \
                          asm(nop);}


#define ExitCritical()  { asm(deca.l SP);\
                             asm(move.l x:(SP)-,SR); \
                             asm(nop);\
                          asm(nop);}


#pragma check_inline_sp_effects on


int func()
{
          int a=1, b=1, c;


          EnterCritical();


          c = a+b;


          ExitCritical();


}
```

This case will work because there are no control flow merge points. SP is modified consistently along all paths from the beginning to the end of the function and is properly aligned.

Listing: Example 2 - Illegal Modification of SP using Inline Assembly

```
#define EnterCritical() { asm(adda #2,SP);\
                          asm(move.l  SR,X:(SP)+); \
                          asm(bfset  #0x0300,SR); \
                          asm(nop); \
                          asm(nop);}

#define ExitCritical()  { asm(deca.l SP);\
                            asm(move.l x:(SP)-,SR); \
                            asm(nop);\
                          asm(nop);}

#pragma check_inline_sp_effects on

int func()
{
          int a=1, b=1, c;

          if (a)
{
                  EnterCritical();

                  c = a+b;

          }
          else {
                  c = b++;
          }

          ExitCritical();

          return (b+c);
}
```

This example will generate the following warning because the SP entering the "ExitCritical" macro is different depending on which branch is taken in the if. Therefore, accesses to variables a, b, or c may not be correct.

```
Warning : Inconsistent inline assembly modification of SP in this function.
M56800E_main.c line 29    ExitCritical();
```

Listing: Example 3 - Modification of SP by a Run-time Dependent Amount

```
#define EnterCritical() { asm(adda R0,SP);\
                          asm(move,l  SR,X:(SP)+); \
                          asm(bfset   #0x0300,SR); \
                          asm(nop); \
                          asm(nop);}

#define ExitCritical()  { asm(deca.l SP);\
                          asm(move.l X:(SP)-,SR); \
                          asm(nop);\
                          asm(nop);}

#pragma check_inline_sp_effects on
int func()
{
          int a=1, b=1, c;

          if (a)
          {
                  EnterCritical();

                  c = a+b;

          }
          else {
                  EnterCritical();
                  c = b++;
          }
```

```
        return (b+c);
}
```

This example will generate the following warning:

```
Warning : Cannot determine SP modification value at compile time
M56800E_main.c line 20     EnterCritical();
```

This example is not legal since the SP is modified by run-time dependent amount.

If all inline assembly modifications to the SP along all branches are equal approaching the exit of a function, it is not necessary to explicitly deallocate the increased stack space. The compiler "cleans up" the extra inline assembly stack allocation automatically at the end of the function.

Listing: Example 4 - Automatic Deallocation of Inline Assembly Stack Allocation

```
#pragma check_inline_sp_effects on
int func()
{
        int a=1, b=1, c;

        if (a)
        {
                EnterCritical();

                c = a+b;

        }
        else {
                EnterCritical();
                c = b++;
        }

        return (b+c);
}
```

This example does not need to call the "ExitCritical" macro because the compiler will automatically clean up the extra inline assembly stack allocation.

4.4 Data Alignment Requirements

The data alignment rules for DSP56800E stack and global memory are:

- Bytes — byte boundaries.
- Words — word boundaries.
- Longs, floats, and doubles — double-word boundaries:
  - Least significant word is always on an even word address.
  - Most significant word is always on an odd word address.
  - Long accesses through pointers in AGU registers (for example, R0 through R5 or N) point to the least significant word. That is, the address is even.
  - Long accesses through pointers using SP point to the most significant word. That is, the address in SP is odd.
- Structures — word boundaries (not byte boundaries).

NOTE

A structure containing only bytes still is word aligned.

- Structures — double-word boundaries if they contain 32-bit elements, or if an inner structure itself is double-word aligned.
- Arrays — the size of one array element.

This topic contains the following sub-topics:

- Word and Byte Pointers
- Reordering Data for Optimal Usage

4.4.1 Word and Byte Pointers

The alignment requirements explained above determine how the compiler uses DSP56800E byte and word pointers to implement C pointer types. The compiler uses:

- Word pointers for all structures
- The SP to access the stack resident data of all types:
  - Bytes
  - Shorts
  - Longs
  - Floats

  - Doubles
  - Any pointer variables
- Word pointers to access:
  - Shorts
  - Longs
  - Any pointer variables
- Byte pointers for:
  - Single global or static byte variable, if accessed through a pointer using X:(Rn)
  - Global or static array of byte variables

    The compiler does not use pointers to access scalar global or static byte variables directly by their addresses. Instead, it uses an instruction with a .BP suffix:

```
MOVE[U].BP    X:xxxx,<dest>

MOVE.BP       <src>,X:xxxx
```

4.4.2 Reordering Data for Optimal Usage

The compiler changes data order, for optimal usage. The data reordering follows these guidelines:

- Reordering is mandatory if local variables are allocated on the stack.
- The compiler does not reorder data for parameter values passed in memory (instead of being passed in registers).
- The compiler does not reorder data when locating fields within a structure.

4.5 Variables in Program Memory

This feature allows the programmer full flexibility in deciding the placement of variables in memory. Variables can be now declared as part of the program memory, using a very simple and intuitive syntax. For example:

```
__pmem int c; // 'c' is an integer that will be stored in program memory.
```

This feature is very useful when data memory is tight, because some or all of the data can be moved to program memory. It can be handled exactly the same way as normal data. This is almost completely transparent to the programmer, with a few exceptions that will be presented in the next paragraphs.

The CPU architecture only allows post increment addressing of words (16-bit data) in program memory. While the compiler circumvents this restriction and allows full access to all data types in program memory, the performance is decreased. If placement of some variables in program memory is needed, and at the same time the execution speed is important, here are some pointers that can be used to organize the code:

- Try to keep all variables that are used in a loop (the loop counter included) in data memory. This condition becomes more important as the loop nesting level increases.
- If possible, place only int (16-bit) data in program memory. Data types with different dimensions are accessed via sequences of code rather than single instructions. 16-bit data is fastest, followed by 32-bit data and 8-bit data.
- Data in program memory can be loaded and stored in a limited number of DALU registers. Because of this, a number of register save/restore sequences can appear if there are not enough available DALU registers. This could be a problem with computational intensive code because the operations do not take place only in registers anymore, and the execution of the code will be slower. This can be avoided by using as many variables in data memory as possible.

This topic contains the following sub-topics:

4.5.1 Declaring Program Memory Variables

A program memory variable is declared using the `__pmem` qualifier. Here are some examples:

```
typedef struct // simple structure declaration
{
   int i;
   char *p;
   long l;
} test;


__pmem int ip1 = 5; // initialized int in program memory
__pmem int ip2; // uninitialized int in program memory
int *__pmem ppx1; // pointer in program memory to int in data memory
__pmem int * __pmem ppp1; // pointer in program memory to int in program memory
__pmem int parr[ 100 ]; // array in program memory
```

```
__pmem test sp; // structure in program memory

__pmem int aap[ 2 ][ 2 ]; // two dimensional array in program memory

__pmem int *pxp1; // pointer in data memory to int in program memory
```

4.5.2 Using Variables in Program Memory

Variables in program memory can be used almost exactly like variables in data memory. The exceptions are presented below:

- The __pmem qualifier cannot be used in a structure declaration because a structure can have all its members either in program memory or in data memory, but not in both memory spaces. The compiler will issue an error message in this case. For example:

```
typedef struct // simple structure declaration

{

   int i;

   char __pmem *p; // error, __pmem not allowed here

   long l;

} test;
```

- The compiler will signal an error when an implicit conversion between a pointer to data in data memory and a pointer to data in program memory is attempted. For example, using the previous definitions, the compiler gives an error for this assignment:

```
pxp1 = ppx1;
```

Explicit conversions are allowed, but they should be used with care. An explicit conversion for the previous assignment that is accepted by the compiler is given below:

```
pxp1 = ( __pmem int * )ppx1;
```

Another consequence of this restriction is that an important part of the MSL functions that have at least an argument that is a pointer will not work with variables in program memory. For example:

```
char *c1; // pointer in data memory to char in data memory

char __pmem *c2; // pointer in data memory to char in program
```

```
memoryvstrcat( c1, c2 ); // error, the second argument can't be converted to 'const char *'
```

If variable argument lists are used, this problem is generally hidden. The program is compiled with no errors from the compiler, but it doesn't work as expected. The most common example is the `printf` function:

```
char *c1 = "xmem"; // pointer in data memory to char in data memory
char __pmem *c2 = "pmem"; // pointer in data memory to char in program memory

printf( "%s\n", c1 );     // works as expected
printf( "%s\n", c2 );     // doesn't work as expected
```

Here, the type of the arguments is lost because `printf` uses a variable argument list. Thus the compiler can not signal a type mismatch and the program will compile without errors, but it won't work as expected, because `printf` assumes that all the data is stored in data memory.

4.5.3 Linking with Variables in Program Memory

The compiler creates special sections in the output file for variables in program memory.

This is a description of all data in program memory sections:

- .data.pmem (initialized program memory data)
- .const.data.pmem (constant program memory data)
- bss.pmem (uninitialized program memory data)

The following sections are also generated if you choose to generate separate sections for char data:

- .data.char.pmem (initialized program memory chars)
- .const.data.char.pmem (constant program memory chars)
- .bss.char.pmem (uninitialized program memory chars)

These sections are used in the linker command file just like normal sections. A typical linker command file for a program that uses data in program memory looks like the following listing

NOTE

`__pmem` qualifier can be used only for global variable and is not available for local variable.

Listing: Typical Linker Command File

```
MEMORY

{

    .p_RAM              (RWX) : ORIGIN = 0x0082,   LENGTH = 0xFF3E

    .p_reserved_regs    (RWX) : ORIGIN = 0xFFC0,   LENGTH = 0x003F

    .p_RAM2             (RWX) : ORIGIN = 0xFFFF,   LENGTH = 0x0000

    .x_RAM              (RW)  : ORIGIN = 0x0001,   LENGTH = 0x7FFE   # SDM xRAM limit is
0x7FFF

}

SECTIONS

{

    .application_code :

    {v        # .text sections


        * (.text)

        * (rtlib.text)

        * (fp_engine.text)

        * (user.text)

        * (.data.pmem)            # program memory initalized data

        * (.const.data.pmem)      # program memory constant data

        * (.bss.pmem)             # program memory uninitialized data

    } > .p_RAM


    .data :

    {

        # .data sections


        * (.const.data.char)  # used if "Emit Separate Char Data Section" enabled

        * (.const.data)v        * (fp_state.data)

        * (rtlib.data)

        * (.data.char)        # used if "Emit Separate Char Data Section" enabled

        * (.data)


        # .bss sections
```

```
        * (rtlib.bss.lo)
        * (rtlib.bss)
        . = ALIGN(1);
        _START_BSS = .;
        * (.bss.char)         # used if "Emit Separate Char Data Section" enabled
        * (.bss)
        _END_BSS   = .;

        # setup the heap address

        . = ALIGN(4);
        _HEAP_ADDR = .;
        _HEAP_SIZE = 0x100;
        _HEAP_END = _HEAP_ADDR + _HEAP_SIZE;
        . = _HEAP_END;

        # setup the stack address

        _min_stack_size = 0x200;
        _stack_addr = _HEAP_END;
        _stack_end  = _stack_addr + _min_stack_size;
        . = _stack_end;

        # export heap and stack runtime to libraries

        F_heap_addr   = _HEAP_ADDR;
        F_heap_end    = _HEAP_END;
        F_Lstack_addr = _HEAP_END;
        F_start_bss   = _START_BSS;
        F_end_bss     = _END_BSS;
    } > .x_RAM
}
```

4.6 Code and Data Storage

The DSP56800E processor has a dual Harvard architecture with separate CODE (P: memory) and DATA (X: memory) memory spaces. The following table shows the sizes and ranges of these spaces, as well as the range of character data within X memory, for both the small and large memory models. (You may need to use the **ELF Linker and Command Language** or **M56800E Linker settings** panel to specify how the project-defined sections map to real memory.)

Table 4-4. Code and Data Memory Ranges

Section	Small Model		Large Model	
	Size	**Range** ((Word Address))	**Size**	**Range** ((Word Address))
CODE (P: memory)	128 KB	0 - 0xFFFF	1 MB	0 - 0x7FFFF
DATA (X: memory)	128 KB	0 - 0xFFFF	32 MB	0 - 0xFFFFFF
DATA (X: memory) character data	64 KB	0 - 0x7FFF	16 MB	0 - 0x7FFFFF

A peculiarity of the DSP56800E architecture is byte addresses for character (1-byte) data, but word addresses for data of all other types. To calculate a byte address, multiply the word address by 2. An address cannot exceed the maximum physical address, so placing character data in the upper half of memory makes the data unaddressable. (Address registers have a fixed width.)

For example, in the small memory model (maximum data address: 64 KB), placing character data at 0x8001 requires an access address of 0x10002. But this access address does not fit into 16-bit storage, as the small data memory model requires. Under your control, the compiler increases flexibility by placing all character data into specially-named sections as described in DSC Compiler > Processor. You can locate these sections in the lower half of the memory map, making sure that the data can be addressed.

4.7 Large Data Model Support

The DSP56800E extends the DSP56800 data addressing range, by providing 24-bit address capability to some instructions. 24-bit address modes allow user accesses beyond the 64K-word boundary of 16-bit addressing. To control large data memory model support, use the M56800E Processor panel as shown in the following figure. See DSC Compiler > Processor for explanations of this panel's elements.

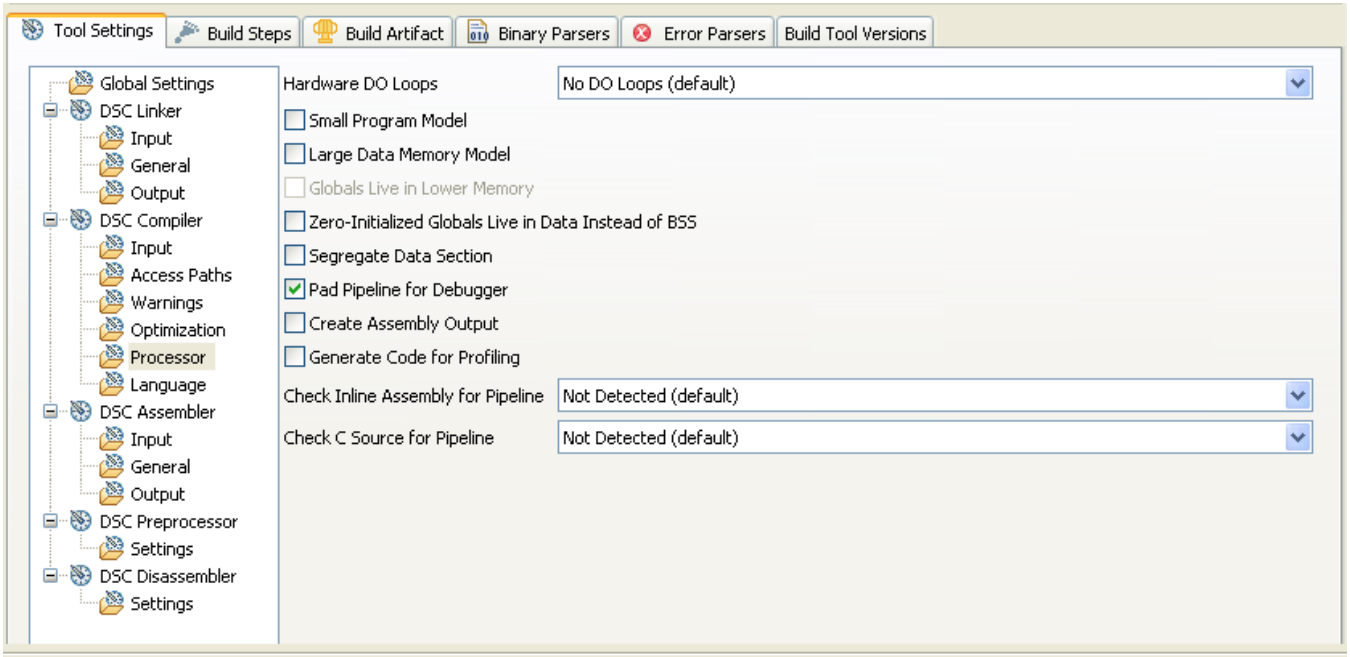

Figure 4-2. M56800E Processor Panel: Large Data Model

Extended data is data located beyond the 16-bit address boundary — as if it exists in extended (upper) memory. Memory located below the 64K boundary is *lower memory*.

The compiler default arrangement is using 16-bit addresses for all data accesses. This means that absolute addresses (X:xxxx addressing mode) are limited to 16 bits. Direct addressing or pointer registers load or store 16-bit addresses. Indexed addressing indexes are 16-bit quantities. The compiler treats data pointers as 16-bit pointers that you may store in single words of memory.

NOTE

There is a known compiler limitation for the default small data model. The data access from 0x8000 to 0xFFFF can result in wrong memory access due to the existing property of 'Index by 16-Bit Displacement: (Rn+xxxx)' - Address-Register-Indirect Addressing Mode. This is because the address plus offset calculation is actually a signed operation in this mode. To overcome this issue, the `-largeAddrInSdm` compiler option should be used, if data access above 0x7fff is required with the default small data model. For details, refer -largeAddrInSdm.

Linker support is present to error out with correct help message for data access above 0x7FFF (without `-largeAddrInSdm` option usage) for small data model cases.

If the large data memory model is enabled, the compiler accesses all data by 24-bit addressing modes. It treats data pointers as 24-bit quantities that you may store in two words of memory. Absolute addressing occurs as 24-bit absolute addresses. Thus, you may access the entire 24-bit data memory, locating data objects anywhere.

You do not need to change C source code to take advantage of the large data memory model.

Examples in DSP56800E assembly code of extended data addressing are:

- Extended Data Addressing Example
- Accessing Data Objects Examples
- External Library Compatibility

4.7.1 Extended Data Addressing Example

Consider the code of the following listing.

Listing: Addressing Extended Data

```
move.w x:0x123456,A1     ; move int using 24 bit absolute address

tst.l   x:(R0-0x123456) ; test a global long for zero using 24-bit pointer indexed

                         ; addressing

move.l r0,x:(R0)+        ; r0 stored as 24-bit quantity

cmpa r0,r1               ; compare pointer registers as 24 bit quantities
```

The large data memory model is convenient because you can place data objects anywhere in the 24-bit data memory map. But the model is inefficient because extended data addressing requires more program memory and additional execution cycles.

However, all global and static data of many target applications easily fit within the 64 K word memory boundary. With this in mind, you can check the **Globals live in lower memory** checkbox of the M56800E Processor settings panel. This tells the compiler to access global and static data with 16-bit addresses, but to use 24-bit addressing for all pointer and stack operations. This arrangement combines the flexibility of the large data memory model with the efficiency of the small data model's access to globals and statics.

NOTE

If you check the Globals live in lower memory checkbox, be sure to store data in lower memory.

4.7.2 Accessing Data Objects Examples

The tables below show appropriate ways to access a global integer and a global pointer variable. The first two columns of each table list states of two checkboxes, Large Data Model and Globals live in lower memory. Both checkboxes are in the M56800E Processor settings panel; the first enables the second.

The following table lists ways to access a global integer stored at address X:0x1234.

```
int g1;
```

Table 4-5. Accessing Global Integer

Large Data Model checkbox	Globals Live in Lower Memory Checkbox	Instruction	Comments
Clear	Clear	move.w X:0x1234,y0	Default values
Checked	Clear	move.w X:0x001234,y0	
Clear	Checked	Combination not allowed	
Checked	Checked	move.w X:0x1234,y0	Global accesses use 16-bit addressing

The following table lists ways to load a global pointer variable, at X:0x4567, into an address register.

```
int * gp1;
```

Table 4-6. Loading Global Pointer Variable

Large Data Model checkbox	Globals Live in Lower Memory Checkbox	Instruction	Comments
Clear	Clear	move.w X:0x4567,r0	Default 16-bit addressing, 16-bit pointer value
Checked	Clear	move.l X:0x004567,r0	24-bit addressing, pointer value is 24-bit
Clear	Checked	Combination not allowed	
Checked	Checked	move.l X:0x4567,r0	16-bit addressing, pointer value is 24-bit

4.7.3 External Library Compatibility

If you enable the large data model when the compiler builds your main application, external libraries written in C also must be built with the large data model enabled. The linker enforces this requirement, catching global objects located out of range for particular instructions.

A more serious compatibility problem involves pointer parameters. Applications built with the large data memory model may pass pointer parameter values in two words of the stack. But libraries built using the small memory model may expect pointer arguments to occupy a single word of memory. This incompatibility will cause runtime stack corruption.

You may or may not build external libraries or modules written in assembly with extended addressing modes. The linker does not enforce any compatibility rules on assembly language modules or libraries.

The compiler encodes the memory model into the object file. The linker verifies that all objects linked into an executable have compatible memory models. The ELF header's e_flags field includes the bit fields that contain the encoded data memory model attributes of the object file:

```
#define EF_M56800E_LDMM 0x00000001 /* Large data memory model flag */
```

Additionally, C language objects are identified by an ELF header flag.

```
#define EF_M56800E_C 0x00000002 /* Object file generated from C source */
```

4.8 Optimizing Code

Register coloring is an optimization specific to DSP56800E development. The compiler assigns two (or more) register variables to the same register, if the code does not use the variables at the same time. The code of the following listing does not use variables i and j at the same time, so the compiler could store them in the same register:

However, if the code included the expression `MyFunc (i+j)`, the variables would be in use at the same time. The compiler would store the two variables in different registers.

For DSP56800E development, you can instruct the compiler to:

1. **Store all local variables on the stack.** — (That is, do *not* perform register coloring.) The compiler loads and stores local variables when you read them and write to them. You may prefer this behavior during debugging, because it guarantees meaningful values for all variables, from initialization through the end of the function. To have

the compiler behave this way, specify **Optimizations Off**, in the **Global Optimizations** settings panel.

2. **Place as many local variables as possible in registers.** — (That is, *do* perform register coloring.) To have the compiler behave this way, specify optimization Level 1 or higher, in the **Global Optimizations** settings panel.

NOTE

Optimizations Off is best for code that you will debug after compilation. Other optimization levels include register coloring. If you compile code with an optimization level greater than 0 and then debug the code, register coloring could produce unexpected results.

Variables declared `volatile` (or those that have the address taken) are not kept in registers and may be useful in the presence of interrupts.

3. **Run Peephole Optimization.** — The compiler eliminates some compare instructions and improves branch sequences. Peephole optimizations are small and local optimizations that eliminate some compare instructions and improve branch sequences. To have the compiler behave this way, specify optimization Levels 1 through 4, in the **Global Optimizations** settings panel.

4.9 Deadstripping and Link Order

The M56800E Linker deadstrips unused code and data only from files compiled by the CodeWarrior C compiler. The linker never deadstrips assembler relocatable files or C object files built by other compilers.

Libraries built with the CodeWarrior C compiler contribute only the used objects to the linked program. If a library has assembly files or files built with other C compilers, the only files that contribute to the linked program are those that have at least one referenced object. If you enable deadstripping, the linker completely ignores files without any referenced objects.

The Link Order page of the project window specifies the order (top to bottom) in which the DSP56800E linker processes C source files, assembly source files, and archive (.a and .lib) files. If both a source-code file and a library file define a symbol, the linker uses the definition of the file that appears first, in the link order. To change the link order, drag the appropriate filename to a different place, in this page's list.

4.10 Working with Peripheral Module Registers

This section highlights the issues and recommends programming style for using bit fields to access memory mapped I/O. Memory mapped I/O is a way of accessing devices that are not on the system. A part of the normal address space is mapped to I/O ports. A read/write to that memory location triggers an access to the I/O device, though to the program it seems like a normal memory access. Even if one byte is written to in the space allocated to a peripheral register, the whole register is written to. So the other byte of the peripheral register will not retain its data. This may happen because the compiler generates optimal bit-field instructions with a read(byte)-mask-writeback(byte) code sequence.

This topic contains the following sub-topics:

4.10.1 Compiler Generates Bit Instructions

The compiler generates BFSET for |= , BFCLR for &=, and BFCHG for ^= operators.

The following listing shows a C source example and the generated sample code.

Listing: C Source Example

```
int i;

int *ip;


void main(void)

{

     i &= ~1;



/* generated codes

P: 00000082:  8054022D0001     bfclr   #1,X:0x00022d

*/


     (*(ip))^= 1;


/* generated codes
```

```
P:00000085: F87C022C            moveu.w  X:0x00022c,R0
P:00000087: 84400001            bfchg     #1,X:(R0)
*/


     *((int*)(0x1234))|=1;


/* generated codes
P:00000089: E4081234            move.l    #4660,R0
P:0000008B: 82400001            bfset     #1,X:(R0)
*/


}


/* generated codes
P:0000008D: E708                rts
*/
```

Note the following example:

```
#define word int
union {
  word Word;
  struct {
    word SBK  :1;
    word RWU  :1;
    word RE   :1;
    word TE   :1;
    word REIE :1;
    word RFIE :1;
    word TIIE :1;
    word TEIE :1;
    word PT   :1;
    word PE   :1;
    word POL  :1;
    word WAKE :1;
    word M    :1;
```

```
    word RSRC :1;
    word SWAI :1;
    word LOOP :1;
  } Bits;
} SCICR;

/* Code:*/
     SCICR.Bits.TE = 1;            /* SCICR content is 0x0800 */
     SCICR.Bits.PE = 1;            /* SCICR content is 0x0002 ??? */
```

4.10.2 Explanation of Undesired Behaviors

If “SCICR” is mapped to a peripheral register, the code that is used to access the register is not portable and might be unsafe, like in DSP56800E at present.

Bit field behavior in C is almost all implementation defined. So generating the following code is legal:

```
     SCICR.Bits.TE = 1;            /* SCICR content is 0x0800 */

/* generated codes
P:00000082:874802c     moveu.w     #SCICR,R0
P:00000084:F0E0000     move.b      X:(R0),A
P:00000086:8350008     bfset       #8,A1
P:00000088:9800        move.b      A1,X:(R0)
*/

     SCICR.Bits.PE = 1;     /* SCICR content is 0x0002 ??? */

/* generated codes
P:00000089:F0E00001    move.b      X:(R0+1),A
P:0000008B:83500002    bfset       #2,A1
P:0000008D:9804        move.b      A1,X:(R0+1)
*/
```

However, since the writes (at P:0x88 and at P:0x8D) are byte instructions and only 16 bits can be written to the SCICR register, the other bytes look as if they are filled with zeros before the SCICR is overwritten.

The use of byte accesses is due to a compiler optimization that tries to generate the smallest possible memory access.

4.10.3 Recommended Programming Style

The use of a union of a member that can hold the whole register (the “Word” member above) and a struct that can access the bits of the register (the “Bits” member above) is a good idea.

What is recommended is to read the whole memory mapped register (using the “Word” union member) into a local instance of the union, do the bit-manipulation on the local, and then write the result as a whole word into the memory mapped register. So the C code would look something like:

```
#define word int


union SCICR_union{
  word Word;
  struct {
    word SBK  :1;
    word RWU  :1;
    word RE   :1;
    word TE   :1;
    word REIE :1;
    word RFIE :1;
    word TIIE :1;
    word TEIE :1;
    word PT   :1;
    word PE   :1;
    word POL  :1;
    word WAKE :1;
    word M    :1;
    word RSRC :1;
    word SWAI :1;
```

```
    word LOOP :1;
  } Bits;
} SCICR;



/* Code: */

 union SCICR_union localSCICR;
 localSCICR.Word = SCICR.Word;

/*  generated codes
P:00000083:F07C022C    move.w      X:#SCICR,A
P:00000085:907F        move.w      A1, X: (SP-1)
*/

 localSCICR.Bits.TE = 1;

/*  generated codes
P:00000086:8AB4FFFF    adda        #-1,SP,R0
P:00000088:F0E00000    move.b      X:(R0),A
P:0000008A:83500008    bfset       #8,A1
P:0000008C:9800        move.b      A1,X: (R0)
*/

 localSCICR.Bits.PE = 1;

/*  generated codes
P:0000008D:F0E00001    move.b      X: (R0+1),A
P:0000008F:83500002    bfset       #2,A1
P:00000091:9804        move.b      A1,x: (R0+1)
*/

 SCICR.Word = localSCICR.Word;

*/ generated codes
```

```
P:00000092:B67F022C     move.w      X:(SP-1),X:#SCICR

*/
```

4.11 Generating MAC Instruction Set

The compiler generates the `imac.l` instruction if the C code performs multiplication on two `long` operands which are casted to `short` type; and the product is added to a `long` type. For example, the following code:

```
short a;

short b;

long c;

.....

long d = c+((long)a*(long)b);

.....
```

generates the following assembly:

```
move.w X:0x000000,Y0 ; Fa

move.w X:0x000000,B ; Fb

move.l X:0x000000,A ; Fc

imac.l B1,Y0,A
```

Chapter 5
C Compiler

The CodeWarrior C programming language closely follows the ISO C Standard (ISO/IEC 9899:1990). CodeWarrior C also has extensions to work more effectively with the target platform and to ensure compatibility with other compilers.

This chapter describes these extensions to the ISO C Standard and implementation-defined behaviors:

- Extensions to Standard C
- Implementation-Defined Behavior

NOTE

For 56800/E Target-specific information, see either *CodeWarrior Development Studio for Freescale 56800/E Digital Signal Controllers: DSP56F80x/DSP56F82x Family Targeting Manual* or *CodeWarrior Development Studio for Freescale 56800/E Digital Signal Controllers: MC56F83xx/DSP5685x Family Targeting Manual.*

5.1 Extensions to Standard C

- Unnamed Arguments in Function Definitions
- C++ Comments
- A # Not Followed by a Macro Argument
- Using an Identifier After #endif
- Using Typecasted Pointers as lvalues
- Inline Functions
- Pascal Calling Conventions
- Character Constants as Integer Values
- Converting Pointers to Types of the Same Size
- Getting Alignment and Type Information at Compile Time
- Arrays of Zero Length in Structures

5.1.1 Unnamed Arguments in Function Definitions

(ISO C, §6.9.1) The C compiler can accept unnamed arguments in a function definition.

Listing: Unnamed Function Arguments

```
void f(int ) {} /* OK if ANSI strict checking is disabled */

void f(int i) {} /* ALWAYS OK */
```

The compiler allows this extension if ANSI strict checking is disabled:

- In the IDE, use the **C/C++ Language Settings** panel's **ANSI Strict** setting
- On the command line, use the compiler's `-ansi strict` option
- In source code, use `#pragma ANSI_strict`

5.1.2 C++ Comments

(ISO C, §6.4.9) The C compiler can accept C++ comments (`//`) in source code. C++ comments consist of anything that follows `//` on a line.

Listing: Example of a C++ Comment

```
a = b; // This is a C++ comment
```

To use this feature, disable the ANSI Strict setting in the IDE options under **DSC Compiler >> Language** panel.

5.1.3 A # Not Followed by a Macro Argument

(ISO C, §6.10.3) The C compiler can accept # tokens that do not appear before arguments in macro definitions.

Listing: Preprocessor Macros Using # Without an Argument

```
#define add1(x) #x #1  // OK, but probably not what you wanted:

 // add1(abc) creates "abc"#1

#define add2(x) #x "2" // OK: add2(abc) creates "abc2"
```

To use this feature, disable the ANSI Strict setting in the IDE options under **DSC Compiler >> Language** panel.

5.1.4 Using an Identifier After #endif

(ISO C, §6.10.1) The C compiler can accept identifier tokens after `#endif` and `#else`. This extension helps you match an `#endif` statement with its corresponding `#if`, `#ifdef`, or `#ifndef` statement, as shown here:

```
#ifdef __MWERKS__

# ifndef __cplusplus

 /*

 * . . .

 */

# endif __cplusplus

#endif __MWERKS__
```

To use this feature, disable the ANSI Strict setting in the Language panel.

Tip

If you enable the ANSI Strict setting (thereby disabling this extension), you can still match your `#ifdef` and `#endif` directives. Simply put the identifiers into comments, as sown in following example:

```
#ifdef __MWERKS__

# ifndef __cplusplus

 /*
```

```
 * . . .

 */

# endif /* __cplusplus */

#endif /* __MWERKS__ */
```

5.1.5 Using Typecasted Pointers as lvalues

The C compiler can accept pointers that are typecasted to other pointer types as lvalues.

Listing: Example of a Typecasted Pointer as an lvalue

```
  char *cp;
  ((long *) cp)++;  /* OK if ANSI Strict is disabled. */
```

To use this feature, disable the ANSI Strict setting in the Language panel.

5.1.6 Inline Functions

As in C++, the CodeWarrior C compiler allows the `inline`, `__inline__`, or `__inline` keyword to appear before a function declaration and definition. An inline keyword specifies to the compiler that it should attempt to replace calls to the function with the function's body.

5.1.7 Pascal Calling Conventions

The CodeWarrior C compiler allows the `pascal` keyword to precede a function declaration and definition. This keyword specifies to the compiler that it should use Pascal calling conventions to call this function.

5.1.8 Character Constants as Integer Values

(ISO C, §6.4.4.4) The C compiler lets you use string literals containing 2 to 8 characters to denote 32-bit integer values. The following table shows the examples.

Table 5-1. Integer Values as Character String Constants

Character Constant	Equivalent Hexadecimal Integer Value
'ABCD'	0x41424344 (32-bit value)
'ABC'	0x00414243 (32-bit value)
'AB'	0x4142 (16-bit value)

You cannot disable this extension, and it has no corresponding pragma or setting in any panel.

NOTE

This feature differs from using multibyte character sets, where a single character requires a data type larger than 1 byte.

5.1.9 Converting Pointers to Types of the Same Size

The C compiler allows the conversion of pointer types to integral data types of the same size in global initializations. Since this type of conversion does not conform to the ANSI C standard, it is only available if the ANSI Strict setting is disabled in the Language panel.

Listing: Converting a Pointer to a Same-sized Integral Type

```
char c;
long arr = (long)&c; // accepted (not ISO C)
```

5.1.10 Getting Alignment and Type Information at Compile Time

The C compiler has two built-in functions that return information about a data type's byte alignment and its data type.

The function call `__builtin_align(`*typeID*`)` returns the byte alignment used for the data type *typeID*. This value depends on the target platform for which the compiler is generating object code.

The function call `__builtin_type(`*typeID*`)` returns an integral value that describes the data type *typeID*. This value depends on the target platform for which the compiler is generating object code.

5.1.11 Arrays of Zero Length in Structures

If you disable the ANSI Strict setting in the Language panel, the compiler lets you specify an array of no length as the last item in a structure. The following listing shows an example. You can define arrays with zero as the index value or with no index value.

Listing: Using Zero-Length Arrays

```
struct listOfLongs {
 long listCount;
 long list[0]; // OK if ANSI Strict is disabled, [] is OK, too.
}
```

5.1.12 The "D" Constant Suffix

When the compiler finds a " D" immediately after a floating point constant value, it treats that value as data of type `double`.

5.1.13 The __typeof__() and typeof() Operators

With the `__typeof__()` operator, the compiler lets you specify the data type of an expression. Listing: Example of __typeof__() and typeof() Operators shows an example.

```
__typeof__(
expression)
```

where *expression* is any valid C expression or data type. Because the compiler translates a `__typeof__()` expression into a data type, you can use this expression wherever a normal type would be specified.

Like the `sizeof()` operator, `__typeof__()` is only evaluated at compile time, not at runtime.

Listing: Example of __typeof__() and typeof() Operators

```
char *cp;
int *ip;
long *lp;
__typeof__(*ip) i; /* equivalent to "int i;" */
__typeof__(*lp) l; /* equivalent to "long l;" */
#pragma gcc_extensions on
typeof(*cp) c; /* equivalent to "char c;" */
```

5.1.14 Specifying Variable Addresses in C

The user can tell the compiler to specify the address of a variable in a C file using the : operator. The constant value following the : operator is the word address of the global variable (i.e., `int OneReg : 0xBCD ;` specifies that the global variable `OneReg` resides at word address `0xBCD`).

NOTE

The Linker does not reserve space for global variables declared with the : operator.

5.2 Implementation-Defined Behavior

The ISO C Standard cannot practically define every possible aspect of a compiler implementation. It does, however, list issues that must be defined by the implementation of the compiler. This section describes aspects of the CodeWarrior C compiler that the ISO C standard refers that are not covered in the rest of this manual:

- Diagnostic Messages
- Identifiers

5.2.1 Diagnostic Messages

(ISO C, §6.3.1) In the CodeWarrior IDE, the CodeWarrior C compiler reports error and warning messages in the **Errors and Warnings** window. On the command-line, the CodeWarrior C compiler reports error and warning messages to the standard error file.

5.2.2 Identifiers

(ISO C, §6.4.2) The CodeWarrior C language allows identifiers to have unlimited length. However, only the first 255 characters are significant for internal and external linkage.

Chapter 6
C++ Compiler

This chapter explains the CodeWarrior implementation of the C++ programming language:

- Features and Limitations
- Implementation-Defined Behavior
- GCC Extensions

6.1 Features and Limitations

The CodeWarrior DSC C++ implementation complies with the ISO/IEC 14882:2003 C++ Standard with the following exceptions:

- No Exception Handling support
- Incomplete support for Runtime Type Information (RTTI)
- Incomplete support for Templates

6.2 Implementation-Defined Behavior

Annex A of the ISO/IEC 14882:2003 C++ Standard lists compiler behaviors that are beyond the scope of the standard, but which must be documented for a compiler implementation. This annex also lists minimum guidelines for these behaviors, although a conforming compiler is not required to meet these minimums.

The CodeWarrior C++ compiler has these implementation quantities listed in the following table, based on the ISO/IEC 14882:2003 C++ Standard, Annex A.

NOTE

The term *unlimited* in the table listed below, means that a behavior is limited only by the processing speed or memory

capacity of the computer on which the CodeWarrior C++ compiler is running.

Table 6-1. Implementation Quantities for C/C++ Compiler (ISO/IEC 14882:2003 C++, §A)

Behavior	Standard Minimum Guideline	CodeWarrior Limit
Nesting levels of compound statements, iteration control structures, and selection control structures	256	Unlimited
Nesting levels of conditional inclusion	256	256
Pointer, array, and function declarators (in any combination) modifying an arithmetic, structure, union, or incomplete type in a declaration	256	Unlimited
Nesting levels of parenthesized expressions within a full expression	256	Unlimited
Number of initial characters in an internal identifier or macro name	1024	Unlimited
Number of initial characters in an external identifier	1024	Unlimited
External identifiers in one translation unit	65536	Unlimited
Identifiers with block scope declared in one block	1024	Unlimited
Macro identifiers simultaneously defined in one translation unit	65536	Unlimited
Parameters in one function definition	256	Unlimited
Arguments in one function call	256	Unlimited
Parameters in one macro definition	256	256
Arguments in one macro invocation	256	256
Characters in one logical source line	65536	Unlimited
Characters in a character string literal or wide string literal (after concatenation)	65536	Unlimited
Size of an object	262144	2 GB
Nesting levels for `# include` files	256	256
`Case` labels for a `switch` statement (excluding those for any nested `switch` statements)	16384	Unlimited
Data members in a single class, structure, or union	16384	Unlimited
Enumeration constants in a single enumeration	4096	Unlimited
Levels of nested class, structure, or union definitions in a single struct-declaration-list	256	Unlimited
Functions registered by `atexit()`	32	64
Direct and indirect base classes	16384	Unlimited
Direct base classes for a single class	1024	Unlimited
Members declared in a single class	4096	Unlimited

Table continues on the next page...

Table 6-1. Implementation Quantities for C/C++ Compiler (ISO/IEC 14882:2003 C++, §A) (continued)

Behavior	Standard Minimum Guideline	CodeWarrior Limit
Final overriding virtual functions in a class, accessible or not	16384	Unlimited
Direct and indirect virtual bases of a class	1024	Unlimited
Static members of a class	1024	Unlimited
Friend declarations in a class	4096	Unlimited
Access control declarations in a class	4096	Unlimited
Member initializers in a constructor definition	6144	Unlimited
Scope qualifications of one identifier	256	Unlimited
Nested external specifications	1024	Unlimited
Template arguments in a template declaration	1024	Unlimited
Recursively nested template instantiations	17	64 (adjustable upto 30000 using #pragma template_depth(<n>))
Handlers per try block	256	Unlimited
Throw specifications on a single function declaration	256	Unlimited

6.3 GCC Extensions

The CodeWarrior C++ compiler recognizes some extensions to the ISO/IEC 14882-2003 C++ standard that are also recognized by the GCC (GNU Compiler Collection) C++ compiler.

The compiler allows the use of the :: operator, of the form *class*::*member*, in a class declaration.

Listing: Using :: Operator in Class Declarations

```
class MyClass {

   int MyClass::getval();

};
```

Chapter 7
ELF Linker

The CodeWarrior Executable and Linking Format (ELF) Linker makes a program file out of the object files of your project. The linker also allows you to manipulate code in different ways. You can define variables during linking, control the link order to the granularity of a single function, change the alignment, and even compress code and data segments so that they occupy less space in the output file.

All of these functions are accessed through commands in the linker command file (LCF). The linker command file has its own language complete with keywords, directives, and expressions, that are used to create the specifications for your output code. The syntax and structure of the linker command file is similar to that of a programming language.

This chapter includes the following sections:

7.1 Structure of Linker Command Files

Linker command files contain three main segments:

A command file must contain a memory segment and a sections segment. Closure segments are optional.

7.1.1 Memory Segment

In the memory segment, available memory is divided into segments. The memory segment format looks like the following listing.

Listing: Sample MEMORY segment

```
MEMORY {
segment_1 (RWX): ORIGIN = 0x8000, LENGTH = 0x1000
segment_2 (RWX): ORIGIN = AFTER(segment_1), LENGTH = 0
segment_3 (RWX): ORIGIN = 0x4000, LENGTH = 0x1000, INITVAL = 0xABCD
 data (RW) : ORIGIN = 0x2000, LENGTH = 0x0000
 #segment_name (RW) : ORIGIN = memory address, LENGTH = segment
 #length
 #and so on...
 }
```

The first memory segment definition (`segment_1`) can be broken down as follows:

- the (`RWX`) portion of the segment definition pertains to the ELF access permission of the segment. The (RWX) flags imply **r** ead, **w** rite, and e **x** ecute access.
- `ORIGIN` represents the start address of the memory segment (in this case `0x8000`).
- `LENGTH` represents the size of the memory segment (in this case `0x1000`).
- `INITVAL` represents the link-time initialization value to be used for watermarking a memory segment . For any ` `INITVAL` = (expression)' the `(expression)' is treated as a word value.

  Example

  Assume for above example that there is a program section of length `0xF00` words and it is placed in `segment_ 3`

  ```
  section{
  program_section_0x0F00;
  }>segment_ 3
  ```

  Then, the resulting memory map will have (`0x1000-0xF00=0x100`) words at the end of `segment_3` that will be initialized with the pattern specified in `INITVAL` = `0xABCD` .

Memory segments with RWX attributes are placed into P: memory while RW attributes are placed into X: memory.

If you cannot predict how much space a segment will occupy, you can use the function AFTER and LENGTH = 0 (unlimited length) to fill in the unknown values.

7.1.2 Closure Blocks

The linker is very good at deadstripping unused code and data. Sometimes, however, symbols need to be kept in the output file even if they are never directly referenced. Interrupt handlers, for example, are usually linked at special addresses, without any explicit jumps to transfer control to these places.

Closure blocks provide a way to make symbols immune from deadstripping. The closure is transitive, meaning that symbols referenced by the symbol being closed are also forced into closure, as are any symbols referenced by those symbols, and so on.

NOTE

The closure blocks need to be in place before the SECTIONS definition in the linker command file.

The two types of closure blocks available are:

- Symbol-level

  Use FORCE_ACTIVE to include a symbol into the link that would not be otherwise included. An example is shown in the following listing.

Listing: Sample symbol-level closure block

```
FORCE_ACTIVE {break_handler, interrupt_handler, my_function}
```

- Section-level

  Use KEEP_SECTION when you want to keep a section (usually a user-defined section) in the link. The following listing shows an example.

Listing: Sample section-level closure block

```
KEEP_SECTION {.interrupt1, .interrupt2}
```

A variant is REF_INCLUDE. It keeps a section in the link, but only if the file where it is coming from is referenced. This is very useful to include version numbers. The following listing shows an example of this.

Listing: Sample section-level closure block with file dependency

```
REF_INCLUDE {.version}
```

7.1.3 Sections Segment

Inside the sections segment, you define the contents of your memory segments, and define any global symbols to be used in the output file.

The format of a typical sections block looks like the following listing.

NOTE

As shown in the following listing, the `.bss` section always needs to be put at the end of a segment or in a standalone segment, because it is not a loadable section.

Listing: Sample SECTIONS segment

```
SECTIONS {
   .section_name : #the section name is for your reference
   {               #the section name must begin with a '.'
     filename.c  (.text) #put the .text section from filename.c
     filename2.c (.text) #then the .text section from filename2.c
     filename.c  (.data)
     filename2.c (.data)
     filename.c  (.bss)
     filename2.c (.bss)
      . = ALIGN (0x10);  #align next section on 16-byte boundary.
   } > segment_1    #this means "map these contents to segment_1"

   .next_section_name:
   {
      more content descriptions
   } > segment_x      # end of .next_section_name definition
   }                     # end of the sections block
```

7.2 Linker Command File Syntax

This section explains some practical ways in which to use the commands of the linker command file to perform common tasks.

7.2.1 Alignment

To align data on a specific word-boundary, use the ALIGN and ALIGNALL commands to bump the location counter to the preferred boundary. For example, the following fragment uses `ALIGN` to bump the location counter to the next 16-byte boundary. An example is given in the following listing.

Listing: Sample ALIGN command usage

```
file.c (.text)

. = ALIGN (0x10);

file.c (.data)    # aligned on a word boundary.
```

You can also align data on a specific word-boundary with ALIGNALL, as shown in the following listing.

Listing: Sample ALIGNALL command usage

```
file.c (.text)

ALIGNALL (0x10);  #everything past this point aligned on word boundary

file.c (.data)
```

7.2.2 Arithmetic Operations

Standard C arithmetic and logical operations may be used to define and use symbols in the linker command file. The following table shows the order of precedence for each operator. All operators are left-associative.

Table 7-1. Arithmetic operators

Precedence	Operators
highest (1)	- ~ !
2	* / %
3	+ -
4	>> <<
5	== != > < <= >=
6	&
7	|
8	&&
9	||

NOTE

The shift operator shifts two-bits for each shift operation. The divide operator performs division and rounding.

7.2.3 Comments

Comments may be added by using the pound character (#) or C++ style double-slashes (//). C-style comments are not accepted by the LCF parser. The following listing shows examples of valid comments.

Listing: Sample comments

```
#  This is a one-line comment
* (.text) // This is a partial-line comment
```

7.2.4 Deadstrip Prevention

The M56800E linker removes unused code and data from the output file. This process is called deadstripping. To prevent the linker from deadstripping unreferenced code and data, use the FORCE_ACTIVE, KEEP_SECTION, and REF_INCLUDE directives to preserve them in the output file.

7.2.5 Variables, Expressions, and Integral Types

This section explains variables, expressions, and integral types.

7.2.5.1 Variables and Symbols

All symbol names within a Linker Command File (LCF) start with the underscore character (_), followed by letters, digits, or underscore characters. The following listing shows examples of valid lines for a command file:

Listing: Valid command file lines

```
_dec_num = 99999999;
_hex_num_ = 0x9011276;
```

Variables that are defined within a SECTIONS section can only be used within a SECTIONS section in a linker command file.

7.2.5.1.1 Global Variables

Global variables are accessed in a linker command file with an `F' prepended to the symbol name. This is because the compiler adds an `F' prefix to externally defined symbols.

The following listing shows an example of using a global variable in a linker command file. This example sets the global variable _foot, declared in C with the extern keyword, to the location of the address location current counter.

Listing: Using global variable in LCF

```
F_foot = .;
```

If you use a global symbol in an LCF, as in the above listing, you can access it from C program sources as shown in the following listing.

Listing: Accessing a Global Symbol from C Program Sources

```
extern unsigned long _Lstack_addr[]; int main(void) { unsigned long* StackStartAddr;
StackStartAddr = _Lstack_addr;
```

7.2.5.2 Expressions and Assignments

You can create symbols and assign addresses to those symbols by using the standard assignment operator. An assignment may only be used at the start of an expression, and a semicolon is required at the end of an assignment statement. An example of standard assignment operator usage is shown in the following listing.

Listing: Standard Assignment Operator Usage

```
_symbolicname =
some_expression
;           # Legal

_sym1 + _sym2 = _sym3;          # ILLEGAL!
```

When an expression is evaluated and assigned to a variable, it is given either an absolute or a relocatable type. An absolute expression type is one in which the symbol contains the value that it will have in the output file. A relocatable expression is one in which the value is expressed as a fixed offset from the base of a section.

7.2.5.3 Integral Types

The syntax for linker command file expressions is very similar to the syntax of the C programming language. All integer types are `long` or `unsigned long`.

Octal integers (commonly know as base eight integers) are specified with a leading zero, followed by numeral in the range of zero through seven. The following listing shows valid octal patterns that you can put into your linker command file.

Listing: Sample Octal patterns

```
_octal_number  = 012;

_octal_number2 = 03245;
```

Decimal integers are specified as a non-zero numeral, followed by numerals in the range of zero through nine. To create a negative integer, use the minus sign (-) in front of the number. The following listing shows examples of valid decimal integers that you can write into your linker command file.

Listing: Sample Decimal integers

```
_dec_num       = 9999;
_decimalNumber = -1234;
```

Hexadecimal (base sixteen) integers are specified as `0x` or `0X` (a zero with an X), followed by numerals in the range of zero through nine, and/or characters `A` through `F`. Examples of valid hexadecimal integers that you can put in your linker command file appear in the following listing.

Listing: Sample Hex integers

```
_somenumber       = 0x0F21;
_fudgefactorspace = 0XF00D;
_hexonyou         = 0xcafe;
```

NOTE

When assigning a value to a pointer variable, the value is in byte units despite that in the linked map (.xMAP file), the variable value appears in word units.

7.2.6 File Selection

When defining the contents of a `SECTION` block, specify the source files that are contributing to their sections.

In a large project, the list can become very long. For this reason, you have to use the asterisk (*) keyword. The * keyword represents the filenames of every file in your project. Note that since you have already added the `.text` sections from the `main.c`, `file2.c`, and `file3.c files`, the `*` keyword does not include the `.text` sections from those files again.

7.2.7 Function Selection

The OBJECT keyword allows precise control over how functions are placed within a section. For example, if the functions `pad` and `foot are` to be placed before anything else in a section, use the code as shown in the example in the following listing.

Listing: Sample function selection using OBJECT keyword

```
SECTIONS {
  .program_section :
    {
      OBJECT (Fpad, main.c)
      OBJECT (Ffoot, main.c)
      * (.text)
     } > ROOT
  }
```

NOTE

If an object is written once using the `OBJECT` function selection keyword, the same object will not be written again if you use the '`*`' file selection keyword.

7.2.8 ROM to RAM Copying

In embedded programming, it is common to copy a portion of a program resident in ROM into RAM at runtime. For example, program variables cannot be accessed until they are copied to RAM.

To indicate data or code that is meant to be copied from ROM to RAM, the data or code is assigned two addresses. One address is its resident location in ROM (where it is downloaded). The other is its intended location in RAM (where it is later copied in C code).

Use the `MEMORY` segment to specify the intended RAM location, and the `AT(address)` parameter to specify the resident ROM address.

For example, you have a program and you want to copy all your initialized data into RAM at runtime. The following listing shows the LCF you use to set up for writing data to ROM.

Listing: LCF setup for ROM to RAM copy

```
MEMORY {
  .text (RWX) : ORIGIN = 0x8000, LENGTH = 0x0    # code (p:)
  .data (RW)  : ORIGIN = 0x3000, LENGTH = 0x0    # data (x:)-> RAM
}

SECTIONS{

    .main_application :
  {
     # .text sections

     *(.text)
     *(.rtlib.text)
     *(.fp_engine.txt)
```

```
      *(user.text)
   } > .text

    __ROM_Address = 0x2000
    .data : AT(__ROM_Address)  # ROM Address definition
   {
      # .data sections
      F__Begin_Data = .;       # Start location for RAM (0x3000)
      *(.data)                 # Write data to the section (ROM)
      *(fp_state.data);
      *(rtlib.data);
      F__End_Data = .;         # Get end location for RAM

      # .bss sections
      * (rtlib.bss.lo)
      * (.bss)
      F__ROM_Address = __ROM_Address

} > .data
}
```

To make the runtime copy from ROM to RAM, you need to know where the data starts in ROM (`__ROM_Address`) and the size of the block in ROM you want to copy to RAM. The following listing shows an example to copy all variables in the data section from ROM to RAM in C code.

Listing: ROM to RAM copy from C after writing data flash

```
#include <stdio.h>
#include <string.h>

int GlobalFlash = 6;

// From linker command file
extern __Begin_Data, __ROMAddress, __End_Data;

void main( void )
{
   unsigned short a = 0, b = 0, c = 0;
   unsigned long dataLen  = 0x0;
   unsigned short __myArray[] = { 0xdead, 0xbeef, 0xcafe };

   // Calculate the data length of the X: memory written to Flash
   dataLen = (unsigned long)&__End_Data -
   unsigned long)&__Begin_Data;

   // Block move from ROM to RAM
   memcpy( (unsigned long *)&__Begin_Data,
  (const unsigned long *)&__ROMAddress,dataLen );

   a = GlobalFlash;

   return;
 }
```

7.2.9 Utilizing Program Flash and Data RAM for Constant Data in C

There are many advantages and one disadvantage if constant data in C is flashed to program flash memory (pROM) and copied to data flash memory (xRAM) at startup, with the usual pROM-to-xRAM initialization.

The advantages are:

- constant data is defined and addressed conventionally via C language
- pROM flash space is used for constant data (pROM is usually larger than xROM)
- the pROM flash is now freed up or available

The disadvantage is that the xRAM is consumed for constant data at run-time.

If you wish to store constant data in program flash memory and have it handled by the `pROM-to-xRAM` startup process, a simple change is necessary to the `pROM-to-xRAM` LCF. Simply, place the constant data references into the `data_in_p_flash_ROM` section after the `__xRAM_data_start` variable like the other data references and remove the "data in xROM" section. See the following listing.

Listing: Using typical pROM-to-xRAM LCF

```
.data_in_p_flash_ROM : AT(__pROM_data_start)
 {
   __xRAM_data_start = .;
   * (.const.data.char) # move constant data references here
   * (.const.char)


   * (.data.char)
   * (.data)


   etc.
```

7.2.10 Utilizing Program Flash for User-Defined Constant Section in Assembler

There are many advantages and one disadvantage in writing specific data to pROM with linker commands and accessing this data in assembly,

The advantages are:

- pROM flash space is used for user-specified constant data (pROM is usually larger than xROM), where the constant data is defined and addressed by assembly language
- part of the pROM flash is now freed up or available

The disadvantage is that data is not defined or accessed conventionally via C language; data is specifically flashed to pROM via the linker command file and fetched from pROM with assembly.

If you want to keep specific constant data in pROM and access it from there, you can use the linker commands to explicitly store the data in pROM and then later access the data in pROM with assembly.

The next two sections describe putting data in the pROM flash at build and run-time.

7.2.10.1 Putting Data in pROM Flash at Build-time

The linker commands have specific instructions which set values in the binary image at the build time, as shown in the following listing. For example, WRITEH inserts two bytes of data at the current address of a section. These commands are placed in the LCF, which tells the linker at build time to place data in P or X memory. Optionally, you can also set the current location prior to the write command to ensure a specific location address for easier reference later. The location within the section is not important.

For more information, see the LCF section in this document.

Listing: LCF write example using MC56F832x for build-time

```
.executing_code :

{

   # .text sections


   . = 0x00A4; # optionally set the location -- we use 0x00A4 in this
   case

   WRITEH(0xABCD); # now set some value here; location within the
   section is not important

   * (.text)

   * (interrupt_routines.text)

   * (rtlib.text)

   * (fp_engine.text)

   * (user.text)


   etc
```

```
} > .p_flash_ROM
```

Putting Data in pROM Flash at Run-time

The assembly example in the following listing fetches the pROM-flashed value at run-time in the above listed code.

Listing: LCF write example using MC56F832x for run-time

```
move.l #$00A4, r1 ; move the pROM address into r3
move.w p:(r3)+, x0 ; fetch data from pROM at address r1 into x0
```

7.2.11 Stack and Heap

To reserve space for the stack and heap, arithmetic operations are performed to set the values of the symbols used by the runtime.

The Linker Command File (LCF) performs all the necessary stack and heap initialization. When Stationery is used to create a new project, the appropriate LCFs are added to the new project.

See any Stationery-generated LCFs for examples of how stack and heap are initialized.

7.2.12 Writing Data Directly to Memory

You can write data directly to memory using the `WRITEX` command in the linker command file. The `WRITEB` command writes a byte, the `WRITEH` command writes two bytes, and the `WRITEW` command writes four bytes. You insert the data at the section's current address.

Listing: Embedding data directly into output

```
.example_data_section :
{
   WRITEB 0x48;  //  'H'
   WRITEB 0x69;  //  'i'
   WRITEB 0x21;  //  '!'
}
```

7.3 Linker Command File Keyword Listing

This section explains the keywords available for use when creating CodeWarrior™ Development Studio for 56800/E Digital Signal Controllers application objects with the linker command file. Valid linker command file functions, keywords, directives, and commands are:

7.3.1 . (location counter)

The period character (.) always maintains the current position of the output location. Since the period always refers to a location in a SECTIONS block, it can not be used outside a section definition.

A period may appear anywhere a symbol is allowed. Assigning a value to period that is greater than its current value causes the location counter to move, but the location counter can never be decremented.

This effect can be used to create empty space in an output section. In the example below, the location counter is moved to a position that is 0x1000 words past the symbol FSTART_.

Example

```
.data :
{
     *(.data)
     *(.bss)
     FSTART_ = .;
     . = FSTART_ + 0x1000;
     __end = .;
} > DATA
```

7.3.2 ADDR

The ADDR function returns the address of the named section or memory segment.

Prototype

```
ADDR (sectionName | segmentName | symbol)
```

In the example below, ADDR is used to assign the address of ROOT to the symbol __rootbasecode.

Example

```
 MEMORY{
    ROOT  (RWX) : ORIGIN = 0x8000, LENGTH = 0
}

SECTIONS{
  .code :
  {
  __rootbasecode = ADDR(ROOT);
     *(.text);
  } > ROOT
}
```

NOTE

In order to use segmentName with this command, the segmentName must start with the period character even though segmentNames are not required to start with the period character by the linker, as is the case with sectionName.

7.3.3 ALIGN

The ALIGN function returns the value of the location counter aligned on a boundary specified by the value of alignValue. The alignValue must be a power of two.

Prototype

```
  ALIGN(alignValue)
```

Note that ALIGN does not update the location counter; it only performs arithmetic. To update the location counter, use an assignment such as:

Example

```
. = ALIGN(0x10);     #update location counter to 16
                     #byte alignment
```

7.3.4 ALIGNALL

ALIGNALL is the command version of the ALIGN function. It forces the minimum alignment for all the objects in the current segment to the value of alignValue. The alignValue must be a power of two.

Prototype

```
ALIGNALL(alignValue);
```

Unlike its counterpart ALIGN, ALIGNALL is an actual command. It updates the location counter as each object is written to the output.

Example

```
.code :
{
  ALIGNALL(16);  // Align code on 16 byte boundary
  *     (.init)
  *     (.text)

  ALIGNALL(16);  //align data on 16 byte boundary
  *     (.rodata)
} > .text
```

7.3.5 FORCE_ACTIVE

The FORCE_ACTIVE directive allows you to specify symbols that you do not want the linker to deadstrip. You must specify the symbol(s) you want to keep before you use the SECTIONS keyword.

Prototype

```
FORCE_ACTIVE{ symbol[, symbol] }
```

7.3.6 INCLUDE

The INCLUDE command let you include a binary file in the output file.

Prototype

```
INCLUDE filename
```

7.3.7 KEEP_SECTION

The KEEP_SECTION directive allows you to specify sections that you do not want the linker to deadstrip. You must specify the section(s) you want to keep before you use the SECTIONS keyword.

Prototype

```
KEEP_SECTION{ sectionType[, sectionType] }
```

7.3.8 MEMORY

The MEMORY directive allows you to describe the location and size of memory segment blocks in the target. This directive specifies the linker the memory areas to avoid, and the memory areas into which it links the code and date.

The linker command file may only contain one MEMORY directive. However, within the confines of the MEMORY directive, you may define as many memory segments as you wish.

Prototype

```
MEMORY { memory_spec }
```

The memory_spec is:

segmentName (*accessFlags*) : ORIGIN = *address,* LENGTH = *length,* [COMPRESS] [> *fileName*]

segmentName can include alphanumeric characters and underscore '_' characters.

accessFlags are passed into the output ELF file (Phdr.p_flags). The accessFlags can be:

- R-read
- W-write
- X-executable (for P: memory placement)

ORIGIN address is one of the following:

Table 7-2. Origin Address

A memory address	Specify a hex address, such as 0x8000.
An AFTER command	Use the AFTER(name [,name]) command to tell the linker to place the memory segment after the specified segment. In the example below, overlay1 and overlay2 are placed after the code segment. When multiple memory segments are specified as parameters for AFTER, the highest memory address is used.

Example

```
memory{

code     (RWX) : ORIGIN = 0x8000, LENGTH = 0

overlay1 (RWX) : ORIGIN = AFTER(code), LENGTH = 0

overlay2 (RWX) : ORIGIN = AFTER(code), LENGTH = 0

data     (RW) : ORIGIN = 0x1000, LENGTH = 0

}
```

ORIGIN is the assigned address.

LENGTH is one of the following:

Table 7-3. Length

A value greater than zero	If you try to put more code and data into a memory segment than your specified length allows, the linker stops with an error.
Autolength by specifying zero	When the length is 0, the linker lets you put as much code and data into a memory segment as you want.

NOTE

There is no overflow checking with autolength. The linker can produce an unexpected result if you use the autolength feature without leaving enough free memory space to contain the memory segment. For this reason, when you use autolength, use the AFTER keyword to specify origin addresses.

> fileName is an option to write the segment to a binary file on disk instead of an ELF program header. The binary file is put in the same folder as the ELF output file. This option has two variants:

Table 7-4. Option Choices

>fileName	Writes the segment to a new file.
>>fileName	Appends the segment to an existing file.

7.3.9 OBJECT

The OBJECT keyword allows control over the order in which functions are placed in the output file.

Prototype

```
OBJECT (function, sourcefile.c)
```

It is important to note that if you write an object to the output file using the OBJECT keyword, the same object will not be written again by either the GROUP keyword or the '*' wildcard.

7.3.10 REF_INCLUDE

The REF_INCLUDE directive allows you to specify sections that you do not want the linker to deadstrip, but only if they satisfy a certain condition: the file that contains the section must be referenced. This is useful if you want to include version information from your source file components. You must specify the section(s) you want to keep before you use the SECTIONS keyword.

Prototype

```
REF_INCLUDE{ sectionType [, sectionType]}
```

7.3.11 SECTIONS

A basic SECTIONS directive has the following form:

Prototype

```
SECTIONS { <section_spec> }
```

section_spec is one of the following:

- *sectionName*: [AT (*loadAddress*)] {contents} > *segmentName*
- *sectionName*: [AT (*loadAddress*]] {contents} >> *segmentName*

sectionName is the section name for the output section. It must start with a period character. For example, ".mysection".

AT (loadAddress) is an optional parameter that specifies the address of the section. The default (if not specified) is to make the load address the same as the relocation address.

contents are made up of statements. These statements can:

- Assign a value to a symbol.
- Describe the placement of an output section, including which input sections are placed into it.

segmentName is the predefined memory segment into which you want to put the contents of the section. The two variants are:

Table 7-5. Option Choices

>segmentName	Places the section contents at the beginning of the memory segment segmentName.
>>segmentName	Appends the section contents to the memory segment segmentName.

Example

```
SECTIONS {
   .text : {
           F_textSegmentStart = .;
           footpad.c (.text)
           . = ALIGN (0x10);
           padfoot.c (.text)
           F_textSegmentEnd = .;
   } > TEXT
  .data : { *(.data) } > DATA
  .bss  : { *(.bss) > BSS
  *(COMMON)
  }
}
```

7.3.12 SIZEOF

The SIZEOF function returns the size of the given segment or section. The return value is the size in bytes.

Prototype

```
SIZEOF(sectionName | segmentName | symbol)
```

NOTE

In order to use segmentName with this command, the segmentName must start with the period character even though

segmentNames are not required to start with the period character by the linker, as is the case with sectionName.

7.3.13 SIZEOFW

The `SIZEOFW` function returns the size of the given segment or section. The return value is the size in words.

Prototype

```
SIZEOFW(sectionName | segmentName | symbol)
```

NOTE

In order to use segmentName with this command, the segmentName must start with the period character even though segmentNames are not required to start with the period character by the linker, as is the case with sectionName.

7.3.14 WRITEB

The WRITEB command inserts a byte of data at the current address of a section.

Prototype

```
WRITEB (expression);
```

`expression` is any expression that returns a value `0x00` to `0xFF`.

7.3.15 WRITEH

The WRITEH command inserts two bytes of data at the current address of a section.

Prototype

```
WRITEH (expression);
```

`expression` is any expression that returns a value `0x0000` to `0xFFFF`.

7.3.16 WRITEW

The WRITEW command inserts 4 bytes of data at the current address of a section.

Prototype

```
WRITEW (expression);
```

`expression` is any expression that returns a value `0x00000000` to `0xFFFFFFFF`.

7.4 Command-Line Linker Options

The command-line linker options are:

- -dis[assemble]
- -defaults
- -L+
- -lr
- -l+
- -nofail
- -reverselibsearchpath
- -stdlib
- -S

7.4.1 -dis[assemble]

Disassembles object code and does not link.

Syntax

```
-dis[assemble]
```

Remarks

This option is global and implies `-nostdlib`. See -stdlib.

7.4.2 -defaults

Same as -stdlib.

Syntax

```
-defaults
-nodefaults
```

Remarks

This option is global.

7.4.3 -L+

Adds library search path; searches the current working directory and then system directories. The search paths have global scope over the command line and are searched in the given order.

Syntax

```
-L+path
-l path
```

The parameters are:

`path`

The search path to append.

Remarks

This option is global and case-sensitive.

7.4.4 -lr

Adds recursive library search path; searches the current working directory and then system directories. The search paths have global scope over the command line and are searched in the given order.

Syntax

```
-lr path
```

The parameters are:

`path`

The recursive library search path to append.

Remarks

This option is global.

7.4.5 -l+

Adds a library by searching access paths for a specified library filename.

Syntax

```
-l+file
```

The parameters are:

`file`

Name of the library path to search.

Remarks

The linker searches access path for the specified `lib<file>.<ext>` where `<ext>` is a typical library extension. If the file is not found then the linker searches for `<file>` directly. This option is case-sensitive.

7.4.6 -nofail

Continues importing or disassembling after getting error messages in earlier files.

Syntax

```
-nofail
```

7.4.7 -reverselibsearchpath

Searches in reverse order of library paths.

Syntax

```
-reverselibsearchpath
```

Remarks

This option is global.

7.4.8 -stdlib

Uses system library access paths specified by the environment variable `%MWLibraries%` to add system libraries specified by the environment variable `%MWLibraryFiles%` at the end of link order.

Syntax

```
-stdlib
```

```
-nostdlib
```

Remarks

This option is global.

7.4.9 -S

Disassembles all files and sends output to a file; it does not link. It is same as `-nostdlib`. See -stdlib.

Syntax

```
-S
```

Remarks

This option is global and case-sensitive.

7.5 ELF Linker Options

The ELF linker options are:

- -dead[strip]
- -force_active
- -keep[local]
- -m[ain]
- -map
- -sortbyaddr
- -srec
- -sreceol
- -sreclength
- -usebyteaddr
- -V3

7.5.1 -dead[strip]

Enables dead-stripping of unused code.

Syntax

```
-[no]dead[strip]
```

7.5.2 -force_active

Specifies a list of symbols as undefined; useful in force linking of static libraries.

Syntax

```
-force_active symbol[,...]
```

7.5.3 -keep[local]

Keeps local symbols, such as relocations and output segment names generated during link.

Syntax

```
-keep[local] on|off
```

Remarks

Default setting is `on`.

7.5.4 -m[ain]

Sets main entry point for the application or shared library. Use -main "" to specify no entry point.

Syntax

```
-m[ain] symbol
```

Remarks

The maximum length for `symbol` is 63 chars; default is `FSTART_`.

7.5.5 -map

Generates link map file.

Syntax

```
-map [keyword[,...]]
```

The arguments of `keyword` are:

`closure`

To calculate symbol closures.

`unused`

To list unused symbols.

`showbyte`

To show byte relocation used on symbols.

7.5.6 -sortbyaddr

Sorts S-records by address.

Syntax

```
-sortbyaddr
```

Remarks

See -srec.

7.5.7 -srec

Generates an S-record file; not used for generating static libraries.

Syntax

```
-srec
```

7.5.8 -sreceol

Sets the end-of-line separator for S-record file.

Syntax

```
-sreceol keyword
```

The arguments of `keyword` are:

`mac`

Use Mac OS®-style (\r) end-of-line format.

`dos`

Use Microsoft® Windows®-style (\r\n) end-of-line format. This is the default choice.

`unix`

Use a UNIX-style (\n) end-of-line format.

Remarks

See -srec.

7.5.9 -sreclength

Specifies the length of S-records.

Syntax

```
-sreclength length
```

Remarks

The length size should be a multiple of 4. The value of `length` ranges from 8 to 252. The default range is 64.

7.5.10 -usebyteaddr

Uses byte address in S-record file.

Syntax

```
-usebyteaddr
```

Remarks

See -srec.

7.5.11 -V3

Generates an elf file for 56800EX digital signal controller.

Syntax

```
-V3
```

7.6 Project Options

The DSP project for linker options are:

- -application
- -library

7.6.1 -application

Generates an application.

Syntax

```
-application
```

Remarks

This option is global.

7.6.2 -library

Generates a static library.

Syntax

```
-library
```

Remarks

This option is global.

7.7 Linker C/C++ Support Options

The linker C/C++ support options are:

- -Cpp_exceptions
- -dialect | -lang

7.7.1 -Cpp_exceptions

Enables or disables C++ exceptions.

Syntax

```
-Cpp_exceptions on|off
```

Remarks

The default option in `on`.

7.7.2 -dialect | -lang

Specifies the source language.

Syntax

```
-dialect | -lang keyword
```

The arguments of `keyword` are:

`C`

Considers source as C++ unless its extension is `.c`, `.h`, or `.pch`. This is the default.

`C++`

Considers source as C++ always.

7.8 Errors and Warnings Options

The errors and warnings options are:

- -w[arn[ings]

7.8.1 -w[arn[ings]

Specifies which warnings the linker command-line tool issues. This command is global.

Syntax

```
-w[arn[ings]] keyword[,...]
```

The options of *keyword* are:

`off`

To turn off all warnings.

`on`

To turn on all warnings.

`[no]cmdline`

To issue command-line parser warnings.

`[no]err[or] | [no]iserr[or]`

To treat warnings as errors.

`noSymRedef`

To suppress Symbol Redefined warnings.

`display|dump`

To display a list of active warnings.

7.9 ELF Disassembler Options

The ELF disassembler options are used along with `-S` or `-dis` commands. The ELF disassembler options are:

- -show
- -dispaths

7.9.1 -show

Specifies the information to list in a disassembly.

Syntax

```
-show keyword[,...]
```

The choices for `keyword` are:

`only | none`

Shows no disassembly. Begin a list of choices with `only` or `none` to prevent default information from appearing in the disassembly.

`all`

Shows binary, executable code, detailed, data, extended, and exception information in the disassembly.

`code | nocode`

Shows or does not show executable code sections.

`text | notext`

Equivalent to the `code` and `nocode` choices, respectively.

`comments | nocomments`

Shows or does not show comment field in code. The default is `comments`. This option also shows code sections.

`extended | noextended`

Shows or does not show extended mnemonics. The default is `extended`. This option also shows code sections.

`data | nodata`

Shows hex dumps of sections with `-show verbose`. The default is `data`.

`[no]debug | [no]sym`

Shows symbolics information.

`exceptions | noexceptions`

Shows or does not show exception tables. The default is `exceptions`. This option also shows data sections.

`headers | noheaders`

Shows or does not show ELF headers. The default is `headers`.

`hex | nohex`

Shows or does not show addresses and opcodes in code disassembly. The default is `hex`.

`names | nonames`

Shows or does not show symbol table. The default is `names`.

`relocs | norelocs`

Shows or does not show resolved relocations in code and relocation tables. The default is `relocs`.

`source | nosource`

Shows source in disassembly with -show verbose. It displays entire source file in output else shows only four lines around each function. The default is `source`.

`xtables | noxtables`

Shows or does not show exception tables. The default is `xtables`.

`verbose | noverbose`

Shows or does not show verbose information including hex dump of program segments in the applications. The default is `verbose`.

7.9.2 -dispaths

Used for disassembler file paths mapping; useful in mapping libraries sources.

Syntax

```
-dispaths="src"="dest"
```

Chapter 8
Inline Assembly Language and Intrinsics

The CodeWarrior compiler supports inline assembly language and intrinsic functions using the 56800E and 56800EX instructions. This chapter explains the implementation of Freescale assembly language, with regard to DSP56800E/EX development. It also explains the relevant intrinsic functions.

NOTE

Inline assembly support for 56800EX instructions is provided in compiler for all 32/64 Integer and Fractional instructions, that is Multiply, MAC, and BFSC instruction. The -V3 compiler option is used to enable the inline assembly support for 56800EX instructions.

This chapter includes these sections:

8.1 Inline Assembly Language

This section explains how to use inline assembly language. It includes these sections:

8.1.1 Inline Assembly Overview

To specify assembly-language interpretation for a block of code in your file, use the `asm` keyword and standard DSP56800E instruction mnemonics.

NOTE

To make sure that the C compiler recognizes the `asm` keyword, you must clear the **ANSI Keywords Only** checkbox of the **Language panel** . Differences in calling conventions mean that you cannot re-use DSP56800 assembly code in the DSP56800E compiler.

The following listing shows how to use the `asm` keyword with braces, to specify that an *entire function* is in assembly language.

Listing: Function-level syntax

```
asm <function header> {
   <assembly instructions>
}
```

The function header can be any valid C function header; the local declarations are any valid C local declarations.

The following listing shows how to use the `asm` keyword with braces, to specify that a block of statements or a single statement is in assembly language.

Listing: Statement-level syntax

```
asm { inline assembly statement


  inline assembly statement
    ...

}
asm {inline assembly statement}
```

The inline assembly statement is any valid assembly-language statement.

The following listing shows how to use the `asm` keyword with parentheses, to specify that a single statement is in assembly language. Note that a semicolon must follow the close parenthesis.

Listing: Alternate single-statement syntax

```
asm (inline assembly statement);
```

NOTE

If you apply the `asm` keyword to one statement or a block of statements *within a function*, you must *not* define local variables within any of the inline-assembly statements.

8.1.2 Assembly Language Quick Guide

Keep these rules in mind as you write assembly language functions:

- Each statement must be a *label* or a *function*.
- A *label* can be any identifier not already declared as a local variable.
- All *labels* must follow the syntax:

```
[LocalLabel:]
```

The following listing illustrates the use of labels.

Listing: Labels in M56800E assembly

```
x1:  add  x0,y1,a

x2:

     add  x0,y1,a

x3   add  x0,y1,a  //ERROR, MISSING COLON
```

- All *instructions* must follow the syntax:

```
( (instruction) [operands] )
```

- Each statement must end with a new line
- Assembly language directives, instructions, and registers are not case-sensitive. The following two statements are the same:

```
add x0,y0

ADD X0,Y0
```

- Comments must have the form of C or C++ comments; they must not begin with the ; or # characters. The following listing shows the valid syntax for comments.
  Listing: Valid comment syntax

```
    move.w    x:(r3),y0     #  ERROR

    add.w     x0,y0         // OK

    move.w    r2,x:(sp)     ; ERROR

    adda    r0,r1,n       /* OK */
```

- To optimize a block of inline assembly source code, use the inline assembly directive `.optimize_iasm on` before the code block. Then use the directive `.optimize_iasm off` at the end of the block. (Omitting `.optimize_iasm off` means that optimizations continue to the end of the function.)

8.1.3 Calling Assembly Language Functions from C Code

You can call assembly language functions from C just as you would call any standard C function, using standard C syntax.

- Calling Inline Assembly Language Functions
- Calling Pure Assembly Language Functions

8.1.3.1 Calling Inline Assembly Language Functions

The following listing demonstrates how to create an inline assembly language function in a C source file. This example adds two 16-bit integers and returns the result.

Notice that you are passing two 16-bit addresses to the `add_int` function. You pick up those addresses in R2 and R3, passing the sum back in Y0.

Listing: Sample code - Creating an inline assembly language function

```
asm int add_int( int * i, int * j )
{
  move.w    x:(r2),y0
  move.w    x:(r3),x0
  add   x0,y0
  // int result returned in y0
  rts
}
```

The following listing shows the C calling statement for this inline-assembly-language function.

Listing: Sample Code - Calling an Inline Assembly Language Function

```
  int x = 4, y = 2;

  y = add_int( &x, &y ); /* Returns 6 */
```

8.1.3.2 Calling Pure Assembly Language Functions

If you want C code to call assembly language files, you must specify a `SECTION` mapping for your code, for appropriate linking. You must also specify a memory space location. Usually, this means that the `ORG` directive specifies code to program memory (P) space.

In the definition of an assembly language function, the `GLOBAL` directive must specify the current-section symbols that need to be accessible by other sections.

The following listing is an example of a complete assembly language function. This function writes two 16-bit integers to program memory. A separate function is required for writing to P: memory, because C pointer variables allow access only to X: data memory.

The first parameter is a short value and the second parameter is the 16-bit address.

Listing: Sample code - Creating an assembly language function

```
                              ;"my_asm.asm"
  SECTION user                ;map to user defined section in CODE
  ORG P:                      ;put the following program in P
                              ;memory

  GLOBAL  Fpmemwrite          ;This symbol is defined within the
                              ;current section and should be
                              ;accessible by all sections
Fpmemwrite:
  MOVE    Y1,R0               ;Set up pointer to address
  NOP                         ;Pipeline delay for R0
  MOVE    Y0,P:(R0)+          ;Write 16-bit value to address
                              ;pointed to by R0 in P: memory and
                              ;post-increment R0
  rts                         ;return to calling function

  ENDSEC                      ;End of section
  END                         ;End of source program
```

The following listing shows the C calling statement for this assembly language function.

Listing: Sample code - Calling an assembly language function from C

```
void pmemwrite( short, short );/* Write a value into P: memory */

void main( void )
{
  // ...other code

  // Write the value given in the first parameter to the address
  // of the second parameter in P: memory
  pmemwrite( (short)0xE9C8, (short)0x0010 );

  // other code...
}
```

8.1.4 Calling Functions from Assembly Language

Assembly language programs can call functions written in either C or assembly language.

- From within assembly language instructions, you can call C functions. For example, if the C function definition is:

  ```
  void foot( void ) {

  /* Do something */

  }
  ```

Your assembly language calling statement is:

```
jsr Ffoot
```

- From within assembly language instructions, you can call assembly language functions. For example, if `pmemwrite` is an assembly language function, the assembly language calling statement is:

```
jsr Fpmemwrite
```

8.2 Intrinsic Functions

This section explains CodeWarrior intrinsic functions. It consists of these sections:

- Implementation
- Fractional Arithmetic
- Intrinsic Functions for Math Support
- Modulo Addressing Intrinsic Functions

8.2.1 Implementation

The CodeWarrior for DSP56800E and DSP56800EX has intrinsic functions to generate inline-assembly-language instructions. These intrinsic functions are a CodeWarrior extension to ANSI C.

Use intrinsic functions to target specific processor instructions. For example:

- Intrinsic functions let you pass in data for specific optimized computations. For example, ANSI C data-representation rules may make certain calculations inefficient, forcing the program to jump to runtime math routines. Such calculations would be coded more efficiently as assembly language instructions and intrinsic functions.
- Intrinsic functions can control small tasks, such as enabling saturation. One method is using inline assembly language syntax, specifying the operation in an `asm` block, every time that the operation is required. But intrinsic functions let you merely set the appropriate bit of the operating mode register.

The IDE implements intrinsic functions as inline C functions in file `intrinsics_56800E.h`, in the MSL directory tree. These inline functions contain mostly inline assembly language code. An example is the `abs_s` intrinsic, defined as:

Listing: Example code - Definition of intrinsic function: abs_s

```
#define    abs_s(a) __abs_s(a)
           /* ABS_S */
inline Word16 __abs_s(register Word16 svar1)
{
/*
 *   Defn: Absolute value of a 16-bit integer or fractional value
 *         returning a 16-bit result.
 *         Returns $7fff for an input of $8000
 *
 *   DSP56800E instruction syntax:  abs FFF
 *              Allowed src regs:  FFF
 *              Allowed dst regs:  (same)
 *
 *   Assumptions: OMR's SA bit was set to 1 at least 3 cycles
 *   before this code.
 */
   asm(abs svar1);
   return svar1;
}
```

8.2.2 Fractional Arithmetic

Many of the intrinsic functions use fractional arithmetic with *implied fractional values*. An implied fractional value is a symbol declared as an integer type, but calculated as a fractional type. Data in a memory location or register can be interpreted as fractional or integer, depending on program needs.

All intrinsic functions that generate multiply or divide instructions perform fractional arithmetic on implied fractional values. (These intrinsic functions are DIV, MPY, MAC, MPYR, and MACR) The relationship between a 16-bit integer and a fractional value is:

Fractional Value = Integer Value / (215)

The relationship between a 32-bit integer and a fractional value is similar:

Fractional Value = Long Integer Value / (231)

The following table shows how 16- and 32-bit values can be interpreted as either fractional or integer values.

Table 8-1. Interpretation of 16- and 32-bit Values

Type	Hex	Integer Value	Fixed-Point Value
short int	0x2000	8192	0.25
short int	0xE000	-8192	-0.25
long int	0x20000000	536870912	0.25
long int	0xE0000000	-536870912	-0.25

NOTE

Intrinsic functions use these two macros, Word16. - A macro for signed short, and Word32. - A macro for signed long.

8.2.3 Intrinsic Functions for Math Support

The following table lists the math intrinsic functions. See Modulo Addressing Intrinsic Functions for explanations of the remaining intrinsic functions.

For the latest information about intrinsic functions, refer to file `intrinsics_56800E.h`.

NOTE

Intrinsics for integers contain `int` in the name. Intrinsics for long long support contain `LL` in the name.

NOTE

To use `long long` intrinsics, you must include the `instrinsics_LL_56800E.h` file. Other intrinsics reside in `intrinsics_56800E.h`.

NOTE

Intrinsics library functions of the 56800EX instructions inline assembly support reside in the `intrinsics_56800EX.h` in the MSL directory tree. The existing `-v3` compiler option is used to enable this inline assembly support for 56800EX instructions.

To view intrinsics library functions of the 56800EX instructions, see Table 8-3.

Table 8-2. Intrinsic Functions for DSP56800E

Category	Function	Category (cont.)	Function (cont.)
Absolute/Negate	abs_s	Multiplication/MAC (continued from previous column)	mult_r
	negate		MULT_R_INT
	L_abs		L_mac
	L_negate		L_MAC_INT
	LL_ABS		L_msu
	LL_NEGATE		L_MSU_INT
Addition/Subtraction	add		L_mult
	sub		L_MULT_INT
	L_add		L_mult_ls
	L_sub		L_MULT_LS_INT
	LL_ADD		LL_LL_MULT_INT
	LL_SUB		LL_MULT_INT
Control	stop		LL_LL_MAC_INT
	wait		LL_MAC_INT
	turn_off_conv_rndg		LL_MSU_INT
	turn_off_sat		LL_LL_MSU_INT
	turn_on_conv_rndg		LL_MULT_LS_INT
	turn_on_sat		LL_LL_MULT
Deposit/Extract	extract_h		LL_MULT
	extract_l		LL_LL_MAC
	L_deposit_h		LL_MAC
	L_deposit_l		LL_MSU
	LL_DEPOSIT_H		LL_LL_MSU
	LL_DEPOSIT_L		LL_MULT_LS
Deposit/Extract (cont.)	LL_EXTRACT_H	Normalization	ffs_s
	LL_EXTRACT_L		norm_s
Division	div_s		ffs_l
	DIV_S_INT		norm_l
	div_s4q	Rounding	round_val
	DIV_S4Q_INT		ROUND_INT
	div_ls		LL_ROUND
	DIV_LS_INT	Shifting	shl
	div_ls4q		shlftNs
	DIV_LS4Q_INT		shlfts
	LL_DIV		shr
	LL_DIV_INT		shr_r
	LL_DIV_S4Q_INT		shrtNs

Table continues on the next page...

Table 8-2. Intrinsic Functions for DSP56800E (continued)

Category	Function	Category (cont.)	Function (cont.)
Multiplication/MAC	mac_r		L_shl
	MAC_R_INT		L_shlftNs
	msu_r		L_shlfts
	MSU_R_INT		L_shr
	mult		L_shr_r
	MULT_INT		L_shrtNs

Table 8-3. Intrinsic Functions for DSP56800EX

Category	Function
Multiplication/MAC (56800EX)	V3_L_mult_int
	V3_L_mac_int
	V3_L_mult
	V3_L_mac
	V3_LL_mult_int
	V3_LL_mult

8.2.3.1 Absolute/Negate

The intrinsic functions of the absolute-value/negate group are:

- abs_s
- negate
- L_abs
- L_negate
- LL_ABS
- LL_NEGATE

8.2.3.1.1 abs_s

Absolute value of a 16-bit integer or fractional value returning a 16-bit result. Returns 0x7FFF for an input of 0x8000.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 abs_s(Word16 svar1)
```

Example

```
int result, s1 = 0xE000;  /*  - 0.25    */
result = abs_s(s1);
// Expected value of result: 0x2000 = 0.25
```

8.2.3.1.2 negate

Negates a 16-bit integer or fractional value returning a 16-bit result. Returns 0x7FFF for an input of 0x8000.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 negate(Word16 svar1)
```

Example

```
int result, s1 = 0xE000;  /*  - 0.25    */
result = negate(s1);
// Expected value of result: 0x2000 = 0.25
```

8.2.3.1.3 L_abs

Absolute value of a 32-bit integer or fractional value returning a 32-bit result. Returns 0x7FFFFFFF for an input of 0x80000000.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_abs(Word32 lvar1)
```

Example

```
long result, l = 0xE0000000;  /*  - 0.25    */
 result = L_abs(s1);

// Expected value of result: 0x20000000 = 0.25
```

8.2.3.1.4 L_negate

Negates a 32-bit integer or fractional value returning a 32-bit result. Returns 0x7FFFFFFF for an input of 0x80000000.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_negate(Word32 lvar1)
```

Example

```
long result, l = 0xE0000000;  /*  - 0.25    */
  result = L_negate(s1);
  // Expected value of result: 0x20000000 = 0.25
```

8.2.3.1.5 LL_ABS

Absolute value of a 64-bit integer or fractional value returning a 64-bit result.

Prototype

```
Word64 __LL_abs(Word64 llvar)
```

Example

```
long long s1 = 0xEDCBA98800000000;
long long result;
result = LL_abs (s1);
// Expected value of result: abs(0xEDCBA98800000000) = 0x1234567800000000
```

8.2.3.1.6 LL_NEGATE

Negates a 64-bit integer or fractional value returning a 64-bit result.

Prototype

```
Word64 __LL_negate(Word64 llvar)
```

Example

```
long long s1 = 0x2345678900000000;
long long result;
result = LL_negate (s1);
// Expected value of result: neg(0x2345678900000000) = 0xDCBA987700000000
```

8.2.3.2 Addition/Subtraction

The intrinsic functions of the addition/subtraction group are:

- add
- sub
- L_add
- L_sub
- LL_ADD
- LL_SUB

8.2.3.2.1 add

Addition of two 16-bit integer or fractional values, returning a 16-bit result.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 add(Word16 src_dst, Word16 src2)
```

Example

```
short s1 = 0x4000;  /* 0.5  */
short s2 = 0x2000;  /* 0.25 */
short result;

result = add(s1,s2);
// Expected value of result: 0x6000 = 0.75
```

8.2.3.2.2 sub

Subtraction of two 16-bit integer or fractional values, returning a 16-bit result.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 sub(Word16 src_dst, Word16 src2)
```

Example

```
short s1 = 0x4000;  /* 0.5   */
short s2 = 0xE000;  /* -0.25 */
short result;

result = sub(s1,s2);
// Expected value of result: 0x6000 = 0.75
```

8.2.3.2.3 L_add

Addition of two 32-bit integer or fractional values, returning a 32-bit result.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_add(Word32 src_dst, Word32 src2)
```

Example

```
long la = 0x40000000;  /* 0.5  */
long lb = 0x20000000;  /* 0.25 */
long result;

result = L_add(la,lb);
// Expected value of result: 0x60000000 = 0.75
```

8.2.3.2.4 L_sub

Subtraction of two 32-bit integer or fractional values, returning a 32-bit result.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_sub(Word32 src_dst, Word32 src2)
```

Example

```
long la = 0x40000000;  /* 0.5   */
long lb = 0xE0000000;  /* -0.25 */
long result;

result = L_sub(la,lb);
// Expected value of result: 0x60000000 = 0.75
```

8.2.3.2.5 LL_ADD

Addition of two 64-bit integer or fractional values, returning a 64-bit result.

Prototype

```
Word64 __LL_add(Word64 src_dst, Word64 src2)
```

Example

```
long long s1 = 0x3579BDEF00000000;
long long s2 = 0xA864213500000000;
long long result;
result = L_add (s1, s2);
// Expected value of result: 0x3579BDEF00000000 + 0xA864213500000000 = 0xDDDDDF2400000000
```

8.2.3.2.6 LL_SUB

Subtraction of two 64-bit integer or fractional values, returning a 64-bit result.

Prototype

```
Word64 __LL_sub(Word64 src_dst, Word64 src2)
```

Example

```
long long s1 = 0x2345678900000000;
long long s2 = 0xDCBA987700000000;
long long result;
result = L_sub (s1, s2);
// Expected value of result: 0x2345678900000000 - 0xDCBA987700000000 = 0x468ACF1200000000
```

8.2.3.3 Control

The intrinsic functions of the control group are:

- stop
- wait
- turn_off_conv_rndg
- turn_off_sat
- turn_on_conv_rndg
- turn_on_sat

8.2.3.3.1 stop

Generates a STOP instruction which places the processor in the low power STOP mode.

Prototype

```
void  stop(void)
```

Usage

```
stop();
```

8.2.3.3.2 wait

Generates a WAIT instruction which places the processor in the low power WAIT mode.

Prototype

```
void  wait(void)
```

Usage

```
wait();
```

8.2.3.3.3 turn_off_conv_rndg

Generates a sequence for disabling convergent rounding by setting the R bit in the OMR register and waiting for the enabling to take effect.

NOTE

If convergent rounding is disabled, the assembler performs twos complement rounding.

Prototype

```
void  turn_off_conv_rndg(void)
```

Usage

```
turn_off_conv_rndg();
```

8.2.3.3.4 turn_off_sat

Generates a sequence for disabling automatic saturation in the MAC Output Limiter by clearing the SA bit in the OMR register and waiting for the disabling to take effect.

Prototype

```
void  turn_off_sat(void)
```

Usage

```
turn_off_sat();
```

8.2.3.3.5 turn_on_conv_rndg

Generates a sequence for enabling convergent rounding by clearing the R bit in the OMR register and waiting for the enabling to take effect.

Prototype

```
void  turn_on_conv_rndg(void)
```

Usage

```
turn_on_conv_rndg();
```

8.2.3.3.6 turn_on_sat

Generates a sequence for enabling automatic saturation in the MAC Output Limiter by setting the SA bit in the OMR register and waiting for the enabling to take effect.

Prototype

```
void  turn_on_sat(void)
```

Usage

```
turn_on_sat();
```

8.2.3.4 Deposit/Extract

The intrinsic functions of the deposit/extract group are:

- extract_h
- extract_l
- L_deposit_h
- L_deposit_l
- LL_DEPOSIT_H
- LL_DEPOSIT_L
- LL_EXTRACT_H
- LL_EXTRACT_L

8.2.3.4.1 extract_h

Extracts the 16 MSBs of a 32-bit integer or fractional value. Returns a 16-bit value. Does not perform saturation. When an accumulator is the destination, zeroes out the LSP portion. Corresponds to *truncation* when applied to fractional values.

Prototype

```
Word16 extract_h(Word32 lsrc)
```

Example

```
long l = 0x87654321;

short result;
```

```
result = extract_h(l);

// Expected value of result: 0x8765
```

8.2.3.4.2 extract_l

Extracts the 16 LSBs of a 32-bit integer or fractional value. Returns a 16-bit value. Does not perform saturation. When an accumulator is the destination, zeroes out the LSP portion.

Prototype

```
  Word16 extract_l(Word32 lsrc)
```

Example

```
long l = 0x87654321;

short result;


result = extract_l(l);

// Expected value of result: 0x4321
```

8.2.3.4.3 L_deposit_h

Deposits the 16-bit integer or fractional value into the upper 16 bits of a 32-bit value, and zeroes out the lower 16 bits of a 32-bit value.

Prototype

```
  Word32 L_deposit_h(Word16 ssrc)
```

Example

```
short s1 = 0x3FFF;

long result;


result = L_deposit_h(s1);

// Expected value of result: 0x3fff0000
```

8.2.3.4.4 L_deposit_l

Deposits the 16-bit integer or fractional value into the lower 16 bits of a 32-bit value, and sign extends the upper 16 bits of a 32-bit value.

Prototype

```
Word32 L_deposit_l(Word16 ssrc)
```

Example

```
short s1 = 0x7FFF;
long result;

result = L_deposit_l(s1);
// Expected value of result: 0x00007FFF
```

8.2.3.4.5 LL_DEPOSIT_H

Deposits the 32-bit integer or fractional value into the upper 32-bits of a 64 bit value, and zeros out the lower 32-bits of a 64-bit value.

Prototype

```
Word64 __LL_deposit_h(Word32 lsrc)
```

Example

```
long s = 0x12341234;
long long result;
result = LL_deposit_h (s);
// Expected value of result: 0x0000000012341234
```

8.2.3.4.6 LL_DEPOSIT_L

Deposits the 32-bit integer or fractional value into the lower 32-bits of a 64 bit value, and sign extends the upper 32-bits of a 64-bit value.

Prototype

```
Word64 __LL_deposit_l(Word32 lsrc)
```

Example

```
long s = 0x12341234;
long long result;
result = LL_deposit_l (s);
// Expected value of result: 0x0000000012341234
```

8.2.3.4.7 LL_EXTRACT_H

Extracts the 32 MSBs of a 64-bit integer or fractional value. Returns a 32-bit value.

Prototype

```
Word32 __LL_extract_h(Word64 llsrc)
```

Example

```
long long s = 0x1234123443214321;
long result;
result = LL_extract_h (s);
// Expected value of result: 0x12341234
```

8.2.3.4.8 LL_EXTRACT_L

Extracts the 32 LSBs of a 64-bit integer or fractional value. Returns a 32-bit value.

Prototype

```
Word32 __LL_extract_l(Word64 llsrc)
```

Example

```
long long s = 0x1234123443214321;

long result;

result = LL_extract_h (s);

// Expected value of result: 0x43214321
```

8.2.3.5 Division

The intrinsic functions of the division group are:

- div_s
- DIV_S_INT
- div_s4q
- DIV_S4Q_INT
- div_ls
- DIV_LS_INT
- div_ls4q
- DIV_LS4Q_INT
- LL_DIV
- LL_DIV_INT
- LL_DIV_S4Q_INT

8.2.3.5.1 div_s

Single quadrant division, that is, both operands are of positive 16-bit fractional values, returning a 16-bit result. If both operands are equal, returns 0x7FFF (occurs naturally).

NOTE

Does not check for division overflow or division by zero.

Prototype

```
Word16 div_s(Word16 s_numerator, Word16 s_denominator)
```

Example

```
short s1=0x2000; /*  0.25 */
short s2=0x4000; /*  0.5  */
short result;


result = div_s(s1,s2);
// Expected value of result: 0.25/0.5 = 0.5 = 0x4000
```

8.2.3.5.2 DIV_S_INT

Single quadrant division (i.e. both operands positive) of two 16-bit integer values, returning a 16-bit result. If both operands are equal, returns $7FFF (occurs naturally).

Prototype

```
Word16 __div_s_int(Word16 s_denominator, Word16 s_numerator)
```

Example

```
int s1 = 0x2000; /* 8192 */
int s2 = 0x0800; /* 2048 */
int result;
result = div_s_int (s1,s2);
// Expected value of result: 8192 / 2048 = 4 = 0x0004
```

8.2.3.5.3 div_s4q

Four quadrant division of two 16-bit fractional values, returning a 16-bit result.

NOTE

Does not check for division overflow or division by zero.

Prototype

```
Word16 div_s4q(Word16 s_numerator, Word16 s_denominator)
```

Example

```
short s1=0xE000;                          /*  -0.25 */
short s2=0xC000;                          /*  -0.5  */
short result;
result = div_s4q(s1,s2);
// Expected value of result: -0.25/-0.5 = 0.5 = 0x4000
```

8.2.3.5.4 DIV_S4Q_INT

Four quadrant division of a 16-bit integer dividend and a 16-bit integer divisor, returning a 16-bit result.

Prototype

```
Word16 __div_s4q_int(Word16 s_numerator, Word16
s_denominator)
```

Example

```
int s1 = 0xE000; /* -8192 */
int s2 = 0x0800; /* 2048 */
int result;
result = div_s4q_int (s1,s2);
// Expected value of result: -8192 / 2048 = -4 = 0xFFFC
```

8.2.3.5.5 div_ls

Single quadrant division, that is, both operands are positive two 16-bit fractional values, returning a 16-bit result. If both operands are equal, returns 0x7FFF (occurs naturally).

NOTE

Does not check for division overflow or division by zero.

Prototype

```
Word16 div_ls(Word32 l_numerator, Word16 s_denominator)
```

Example

```
long  l =0x20000000;/*  0.25 */
short s2=0x4000;/*  0.5  */
short result;

result = div_ls(l,s2);
// Expected value of result: 0.25/0.5 = 0.5 = 0x4000
```

8.2.3.5.6 DIV_LS_INT

Single quadrant division (i.e. both operands positive) of a 32-bit integer dividend and a 16-bit integer divisor, returning a 16-bit result. If both operands are equal, returns $7FFF (occurs naturally).

Prototype

```
Word16 __div_ls_int(Word16 s_denominator, Word32 l_numerator)
```

Example

```
int s1 = 0x2000; /* 8192 */
long s2 = 0x08000000; /* 134217728 */
int result;
result = div_s_int (s1,s2);
// Expected value of result: 134217728 / 8192 = 16384 = 0x4000
```

8.2.3.5.7 div_ls4q

Four quadrant division of a 32-bit fractional dividend and a 16-bit fractional divisor, returning a 16-bit result.

NOTE

Does not check for division overflow or division by zero.

Prototype

```
Word16 div_ls4q(Word32 l_numerator, Word16 s_denominator)
```

Example

```
long  l =0xE0000000;/*  -0.25 */
short s2=0xC000;/*  -0.5  */
short result;

result = div_ls4q(s1,s2);
// Expected value of result: -0.25/-0.5 = 0.5 = 0x4000
```

8.2.3.5.8 DIV_LS4Q_INT

Four quadrant division of a 32-bit integer dividend and a 16-bit integer divisor, returning a 16-bit result.

Prototype

```
Word16 __div_ls4q_int(Word16 s_denominator, Word32
l_numerator)
```

Example

```
int s1 = 0xE000; /* -8192 */
long s2 = 0x08000000; /* 134217728 */
int result;
result = div_ls4q_int (s1,s2);
// Expected value of result: 134217728 / -8192 = -16384 = 0xC000
```

8.2.3.5.9 LL_DIV

Division of one 64-bit fractional value and one 32-bit fractional value, returning a 32-bit result.

Prototype

```
Word32 __LL_div(Word64 s_numerator, Word32 s_denominator)
```

Example

```
ong long s1 = 0x1807E01E00000000;
long s2 = 0x300F0000;
long result;
result = LL_div (s1,s2);
// Expected value of result: 0x1807E01E00000000 / 0x300F0000 = 0x40010000
```

NOTE

The `intrinsics_LL_56800E.h` file must be included.

8.2.3.5.10 LL_DIV_INT

Single quadrant division (i.e. both operands positive) of two 64-bit integer values, returning a 64-bit result.

Prototype

```
Word64 __LL_div_int(Word64 s_numerator, Word64 s_denominator)
```

Example

```
long long s1 = 0x000000001807E01E;
long long s2 = 0x000000000000300F;
long long result;
result = LL_div_int (s1, s2);
// Expected value of result: 0x000000001807E01E / 0x000000000000300F = 0x0000000000008002
```

8.2.3.5.11 LL_DIV_S4Q_INT

Four quadrant division of a 64-bit integer dividend and a 64-bit integer divisor, returning a 64-bit result.

Prototype

```
Word64 __LL_div_s4q_int(Word64 s_denominator, Word64
s_numerator)
```

Example

```
long long s1 = 0x000000001807E01E;
long long s2 = 0x000000000000300F;
long long result;
result = LL_div_s4q_int (s1, s2);
// Expected value of result: 0x000000001807E01E / 0x000000000000300F = 0x0000000000008002
```

8.2.3.6 Multiplication/MAC

The intrinsic functions of the multiplication/MAC group are:

- mac_r
- MAC_R_INT
- msu_r
- MSU_R_INT
- mult
- MULT_INT
- mult_r
- MULT_R_INT
- L_mac
- L_MAC_INT
- L_msu
- L_MSU_INT
- L_mult
- L_MULT_INT
- L_mult_ls
- L_MULT_LS_INT
- LL_LL_MULT_INT
- LL_MULT_INT

- LL_LL_MAC_INT
- LL_MAC_INT
- LL_MSU_INT
- LL_LL_MSU_INT
- LL_MULT_LS_INT
- LL_LL_MULT
- LL_MULT
- LL_LL_MAC
- LL_MAC
- LL_MSU
- LL_LL_MSU
- LL_MULT_LS

8.2.3.6.1 mac_r

Multiply two 16-bit fractional values and add to 32-bit fractional value. Round into a 16-bit result, saturating if necessary. When an accumulator is the destination, zeroes out the LSP portion.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

OMR's R bit was set to 1 at least three cycles before this code, that is, twos complement rounding, not convergent rounding.

Prototype

```
Word16 mac_r(Word32 laccum, Word16 sinp1, Word16 sinp2)
```

Example

```
short s1 = 0xC000;/* - 0.5 */
short s2 = 0x4000;/*   0.5 */
short result;
long Acc = 0x0000FFFF;
result = mac_r(Acc,s1,s2);
// Expected value of result: 0xE001
```

8.2.3.6.2 MAC_R_INT

Multiply two 16-bit integer values and add to 32-bit integer value. Round into a 16-bit result.

Prototype

```
Word16 __mac_r_int(Word32 laccum, Word16 sinp1, Word16 sinp2)
```

Example

```
long s1 = 0x20000000;/* 536870912 */

int s2 = 0x2000;/* 8192 */

int s3 = 0x2000;/* 8192 */

int result;

result = mac_r_int (s1, s2, s3);

// Expected value of result : round(8192 * 8192 + 536870912) = round (603979776) = 9216 =
0x2400
```

8.2.3.6.3 msu_r

Multiply two 16-bit fractional values and subtract this product from a 32-bit fractional value. Round into a 16-bit result, saturating if necessary. When an accumulator is the destination, zeroes out the LSP portion.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

OMR's R bit was set to 1 at least three cycles before this code, that is, twos complement rounding, not convergent rounding.

Prototype

```
Word16 msu_r(Word32 laccum, Word16 sinp1, Word16 sinp2)
```

Example

```
short s1 = 0xC000;/* - 0.5 */

short s2 = 0x4000;/*   0.5 */
```

```
short result;

long Acc = 0x20000000;


result = msu_r(Acc,s1,s2);

// Expected value of result: 0x4000
```

8.2.3.6.4 MSU_R_INT

Multiply two 16-bit integer values and substract this product from a 32-bit integer value. Round into a 16-bit result.

Prototype

```
  Word16 __msu_r_int(Word32 laccum, Word16 sinp1, Word16 sinp2)
```

Example

```
long s1 = 0x20000000;/* 536870912 */

int s2 = 0x2000;/* 8192 */

int s3 = 0x2000;/* 8192 */

int result;

result = msu_r_int (s1, s2, s3);

// Expected value of result : round(536870912 - 8192 * 8192) = round (469762048) = 7168 =
0x1c00
```

8.2.3.6.5 mult

Multiply two 16-bit fractional values and truncate into a 16-bit fractional result. Saturates only for the case of 0x8000 x 0x8000. When an accumulator is the destination, zeroes out the LSP portion.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 mult(Word16 sinp1, Word16 sinp2)
```

Example

```
short s1 = 0x2000;/*  0.25 */
short s2 = 0x2000;/*  0.25 */
short result;

result = mult(s1,s2);
// Expected value of result: 0.625 = 0x0800
```

8.2.3.6.6 MULT_INT

Multiply two 16-bit integer values and truncate into a 16-bit integer result.

Prototype

```
Word16 __mult_int(Word16 sinp1, Word16 sinp2)
```

Example

```
int s1 = 0x2000;/* 8192 */
int s2 = 0x2000;/* 8192 */
int result;
result = mult_int (s1, s2);
// Expected value of result : 8192 * 8192 = high (67108864) = 1024 = 0x0400
```

8.2.3.6.7 mult_r

Multiply two 16-bit fractional values, round into a 16-bit fractional result. Saturates only for the case of 0x8000 x 0x8000. When an accumulator is the destination, zeroes out the LSP portion.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

OMR's R bit was set to 1 at least three cycles before this code, that is, twos complement rounding, not convergent rounding.

Prototype

```
Word16 mult_r(Word16 sinp1, Word16 sinp2)
```

Example

```
short s1 = 0x2000;/*   0.25 */
short s2 = 0x2000;/*   0.25 */
short result;

result = mult_r(s1,s2);
// Expected value of result: 0.0625 = 0x0800
```

8.2.3.6.8 MULT_R_INT

Multiply two 16-bit integer values and round into a 16-bit integer result.

Prototype

```
Word16 __mult_r_int(Word16 sinp1, Word16 sinp2)
```

Example

```
int s1 = 0x2000;/* 8192 */
int s2 = 0x2000;/* 8192 */
int result;
result = mult_int (s1, s2);
// Expected value of result : 8192 * 8192 = round (67108864) = 1024 = 0x0400
```

8.2.3.6.9 L_mac

Multiply two 16-bit fractional values and add to 32-bit fractional value, generating a 32-bit result, saturating if necessary.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_mac(Word32 laccum, Word16 sinp1, Word16 sinp2)
```

Example

```
short s1 = 0xC000;/* - 0.5 */
short s2 = 0x4000;/*   0.5 */
long result, Acc = 0x20000000;/*  0.25 */

result = L_mac(Acc,s1,s2);
// Expected value of result: 0
```

8.2.3.6.10 L_MAC_INT

Multiply two 16-bit integer values and add to 32-bit integer value, generating a 32-bit result.

Prototype

```
Word32 __L_mac_int(Word32 laccum, Word16 sinp1, Word16 sinp2)
```

Example

```
long s1 = 0x20000000;/* 536870912 */
int s2 = 0x2000;/* 8192 */
int s3 = 0x2000;/* 8192 */
long result;
result = L_mac_int (s1, s2, s3);
// Expected value of result: 8192 * 8192 + 536870912 = 603979776 = 0x24000000
```

8.2.3.6.11 L_msu

Multiply two 16-bit fractional values and subtract this product from a 32-bit fractional value, saturating if necessary. Generates a 32-bit result.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_msu(Word32 laccum, Word16 sinp1, Word16 sinp2)
```

Example

```
short s1 = 0xC000;/* - 0.5 */
short s2 = 0xC000;/* - 0.5 */
long result, Acc = 0;

result = L_msu(Acc,s1,s2);
// Expected value of result: 0.25
```

8.2.3.6.12 L_MSU_INT

Multiply two 16-bit integer values and substract this product from a 32-bit integer value. Generates a 32-bit result.

Prototype

```
Word32 __L_msu_int(Word32 laccum, Word16 sinp1, Word16 sinp2)
```

Example

```
long s1 = 0x20000000;/* 536870912 */
int s2 = 0x2000;/* 8192 */
```

```
int s3 = 0x2000;/* 8192 */

long result;

result = L_msu_int (s1, s2, s3);

// Expected value of result : 536870912 - 8192 * 8192 = 469762048 = 0x1c000000
```

8.2.3.6.13 L_mult

Multiply two 16-bit fractional values generating a signed 32-bit fractional result. Saturates only for the case of 0x8000 x 0x8000.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_mult(Word16 sinp1, Word16 sinp2)
```

Example

```
short s1 = 0x2000;/*  0.25 */

short s2 = 0x2000;/*  0.25 */

long result;


result = L_mult(s1,s2);

// Expected value of result: 0.0625 = 0x08000000
```

8.2.3.6.14 L_MULT_INT

Multiply two 16-bit integer values generating a 32-bit integer result.

Prototype

```
inline Word32 __L_mult_int(register Word16 sinp1, Word16 sinp2)
```

Example

```
#include <intrinsics_56800E.h>

...

int s1 = 0x2000;/* 8192 */

int s2 = 0x2000;/* 8192 */

long result;

result = _L_mult_int (s1, s2);

// Expected value of result : 8192 * 8192 = 67108864 = 0x04000000
```

8.2.3.6.15 L_mult_ls

Multiply one 32-bit and one-16-bit fractional value, generating a signed 32-bit fractional result. Saturates only for the case of 0x80000000 x 0x8000.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_mult_ls(Word32 linp1, Word16 sinp2)
```

Example

```
long l1 = 0x20000000;/* 0.25 */

short s2 = 0x2000;/* 0.25 */

long result;


result = L_mult(l1,s2);

// Expected value of result: 0.625 = 0x08000000
```

8.2.3.6.16 L_MULT_LS_INT

Multiply one 32-bit and one 16-bit integer value, generating a signed 32-bit integer result.

Prototype

```
Word32 __L_mult_ls_int(Word32 linp1, Word16 sinp2)
```

Example

```
long s1 = 0x20000000;/* 536870912 */

int s2 = 0x2000;/* 8192 */

long result;

result = L_mult_ls_int (s1, s2);

// Expected value of result : high(8192 * 536870912) = high(4398046511104) = 67108864 =
0x04000000
```

8.2.3.6.17 LL_LL_MULT_INT

Multiply two 64-bit integer values generating a signed 64-bit integer result.

Prototype

```
Word64 __LL_LL_mult_int(Word64 sinp1, Word64 sinp2)
```

Example

```
long long s1 = 0x000000000000A003;

long long s2 = 0x000000000000B005;

long long result;

result = LL_LL_mult_int (s1, s2);

// Expected value of result: 0x000000000000A003 * 0x000000000000B005 = 0x000000006E05300F
```

8.2.3.6.18 LL_MULT_INT

Multiply two 32-bit integer values generating a signed 64-bit integer result.

Prototype

```
Word64 __LL_mult_int(Word32 sinp1, Word32 sinp2)
```

Example

```
long s1 = 0x0000A003;

long s2 = 0x0000B005;

long long result;

result = LL_mult_int (s1, s2);

// Expected value of result: 0x0000A003 * 0x0000B005 = 0x000000006E05300F
```

8.2.3.6.19 LL_LL_MAC_INT

Multiply two 64-bit integer values and add to 64-bit integer value, generating a 64-bit result.

Prototype

```
Word64 __LL_LL_mac_int(Word64 laccum, Word64 sinp1, Word64 sinp2)
```

Example

```
long long s1 = 0x000000000000A003;

long long s2 = 0x000000000000B005;

long long s = 0x00000000D0008000;

long long result;

result = LL_LL_mac_int (s, s1, s2);

// Expected value of result: 0x00000000D0008000 + 0x000000000000A003 * 0x000000000000B005 =
0x00000001305B00F
```

8.2.3.6.20 LL_MAC_INT

Multiply two 32-bit integer values and add to 64-bit integer value, generating a 64-bit result.

Prototype

```
Word64 __LL_mac_int(Word64 laccum, Word32 sinp1, Word32 sinp2)
```

Example

```
long s1 = 0x0000A003;

long s2 = 0x0000B005;

long long s = 0x00000000D0008000;

long long result;

result = LL_mac_int (s, s1, s2);

// Expected value of result: 0x00000000D0008000 + 0x0000A003 * 0x0000B005 = 0x00000001305B00F
```

8.2.3.6.21 LL_MSU_INT

Multiply two 32-bit integer values and subtract this product from a 64-bit integer value. Generates a 64-bit result.

Prototype

```
Word64 __LL_msu_int(Word64 laccum, Word32 sinp1, Word32 sinp2)
```

Example

```
long s1 = 0x0000A003;

long s2 = 0x0000B005;

long long s = 0x00000000D0008000;

long long result;

result = LL_msu_int (s, s1, s2);

// Expected value of result: 0x00000000D0008000 - 0x0000A003 * 0x0000B005 = 0x000000061FB4FF1
```

8.2.3.6.22 LL_LL_MSU_INT

Multiply two 64-bit integer values and subtract this product from a 64-bit integer value. Generates a 64-bit result.

Prototype

```
Word64 __LL_LL_msu_int(Word64 laccum, Word64 sinp1, Word64 sinp2)
```

Example

```
long long s1 = 0x000000000000A003;

long long s2 = 0x000000000000B005;

long long s = 0x00000000D0008000;
```

```
long long result;

result = LL_LL_msu_int (s, s1, s2);

// Expected value of result: 0x00000000D0008000 - 0x000000000000A003 * 0x000000000000B005 =
0x000000061FB4FF1
```

8.2.3.6.23 LL_MULT_LS_INT

Multiply a 64-bit integer value with a 32-bit integer value, generating a 64-bit result.

Prototype

```
Word64 __LL_mult_ls_int(Word64 linp1, Word32 sinp2)
```

Example

```
long long s1 = 0x00000000A0030000;

long s2 = 0x0000B005;

long long result;

result = LL_mult_ls_int (s1, s2);

// Expected value of result: 0x00000000A0030000 * 0x0000B005 = 0x00006E05300F0000
```

8.2.3.6.24 LL_LL_MULT

Multiply two 64-bit fractional values generating a signed 64-bit fractional result.

Prototype

```
Word64 __LL_LL_mult(Word64 sinp1, Word64 sinp2)
```

Example

```
long long s1 = 0x00000000A0030000;

long long s2 = 0x00000000B0050000;

long long result;

result = LL_LL_mult (s1, s2);

// Expected value of result: 0x00000000A0030000 * 0x00000000B0050000 = 0xDC0A601E00000000
```

8.2.3.6.25 LL_MULT

Multiply two 32-bit fractional values generating a signed 64-bit fractional result.

Prototype

```
Word64 __LL_mult(Word32 sinp1, Word32 sinp2)
```

Example

```
long s1 = 0xA0030000;
long s2 = 0xB0050000;
long long result;
result = LL_mult (s1, s2);
// Expected value of result: 0xA0030000 * 0xB0050000 = 0x3BFA601E00000000
```

8.2.3.6.26 LL_LL_MAC

Multiply two 64-bit fractional values and add to 64-bit fractional value, generating a 64-bit result.

Prototype

```
Word64 __LL_LL_mac(Word64 laccum, Word64 sinp1, Word64 sinp2)
```

Example

```
long long s1 = 0x000000000000A003;
long long s2 = 0x000000000000B005;
long long s = 0x00000000D0008000;
long long result;
result = LL_LL_mac (s, s1, s2);
// Expected value of result: 0x00000000D0008000 + 0x000000000000A003 * 0x000000000000B005 =
0x00000001AC0AE01E
```

8.2.3.6.27 LL_MAC

Multiply two 32-bit fractional values and add to 64-bit fractional value, generating a 64-bit result.

Prototype

```
Word64 __LL_mac(Word64 laccum, Word32 sinp1, Word32 sinp2)
```

Example

```
long s1 = 0x0000A003;

long s2 = 0x0000B005;

long long s = 0x00000000D0008000;

long long result;

result = LL_mac (s, s1, s2);

// Expected value of result: 0x00000000D0008000 + 0x0000A003 * 0x0000B005 =
0x00000001AC0AE01E
```

8.2.3.6.28 LL_MSU

Multiply two 32-bit fractional values and subtract this product from a 64-bit fractional value. Generates a 64-bit result.

Prototype

```
Word64 __LL_msu(Word64 laccum, Word32 sinp1, Word32 sinp2)
```

Example

```
long s1 = 0x0000A003;

long s2 = 0x0000B005;

long long s = 0x00000000D0008000;

long long result;

result = LL_msu (s, s1, s2);

// Expected value of result: 0x00000000D0008000 - 0x0000A003 * 0x0000B005 =
0xFFFFFFFFF3F61FE2
```

8.2.3.6.29 LL_LL_MSU

Multiply two 64-bit fractional values and subtract this product from a 64-bit fractional value. Generates a 64-bit result.

Prototype

```
Word64 __LL_LL_msu(Word64 laccum, Word64 sinp1, Word64 sinp2)
```

Example

```
long long s1 = 0x000000000000A003;

long long s2 = 0x000000000000B005;

long long s = 0x00000000D0008000;

long long result;

result = LL_LL_msu (s, s1, s2);

// Expected value of result: 0x00000000D0008000 - 0x000000000000A003 * 0x000000000000B005 =
0xFFFFFFFFF3F61FE2
```

8.2.3.6.30 LL_MULT_LS

Multiply a 64-bit fractional value with a 32-bit fractional value, generating a 64-bit result.

Prototype

```
Word64 __LL_mult_ls(Word64 linp1, Word32 sinp2)
```

Example

```
long long s1 = 0x00000000A0030000;

long long s2 = 0x0000B005;

long long result;

result = LL_LL_msu (s1, s2);

// Expected value of result: 0x00000000A0030000 * 0x0000B005 = 0x0000DC0A601E0000
```

8.2.3.7 Multiplication/MAC (56800EX)

The intrinsic functions of the multiplication/MAC group for 56800EX instructions are:

- V3_L_mult_int

- V3_L_mac_int
- V3_L_mult
- V3_L_mac
- V3_LL_mult_int
- V3_LL_mult

8.2.3.7.1 V3_L_mult_int

Multiply two 32-bit integer values and truncate into a 32-bit integer result (using the IMPY32 instruction).

Prototype

```
Word32 V3_L_mult_int(Word32 slinp1, Word32 slinp2)
```

8.2.3.7.2 V3_L_mac_int

Multiply two 32-bit integer values, and add the lower 32-bits of the product to 32-bit integer value, generating a 32-bit result (using the IMAC32 instruction).

Prototype

```
Word32 V3_L_mac_int(Word32 laccum, Word32 slinp1, Word32 slinp2)
```

8.2.3.7.3 V3_L_mult

Multiply two 32-bit fractional values and truncate into a 32-bit fractional result (using the MPY32 instruction).

Prototype

```
Word32 V3_L_mult(Word32 slinp1, Word32 slinp2)
```

8.2.3.7.4 V3_L_mac

Multiply two 32-bit fractional values, and add the higher 32-bits of the product to 32-bit fractional value, generating a 32-bit result (using the MAC32 instruction).

Prototype

```
Word32 V3_L_mac(Word32 laccum, Word32 slinp1, Word32 slinp2)
```

8.2.3.7.5 V3_LL_mult_int

Multiply two 32-bit integer values generating a signed 64-bit integer result (using the IMPY64 instruction).

NOTE

To enable 64-bit long long data type support, compile the project with `#pragma slld on`.

Prototype

```
Word64 V3_LL_mult_int(Word32 slinp1, Word32 slinp2)
```

8.2.3.7.6 V3_LL_mult

Multiply two 32-bit fractional values generating a signed 64-bit fractional result (using the MPY64 instruction).

NOTE

To enable 64-bit long long data type support, compile the project with `"#pragma slld on"`.

Prototype

```
Word64 V3_LL_mult(Word32 slinp1, Word32 slinp2)
```

8.2.3.8 Normalization

The intrinsic functions of the normalization group are:

- ffs_s
- norm_s
- ffs_l
- norm_l

8.2.3.8.1 ffs_s

Computes the number of left shifts required to normalize a 16-bit value, returning a 16-bit result (finds 1st sign bit). Returns a shift count of 31 for an input of 0x0000.

NOTE

Does not actually normalize the value! Also see the intrinsic norm_s which handles the case where the input == 0x0000 differently.

Prototype

```
Word16 ffs_s(Word16 ssrc)
```

Example

```
short s1 = 0x2000;/* .25 */
short result;


result = ffs_s(s1);
// Expected value of result: 1
```

8.2.3.8.2 norm_s

Computes the number of left shifts required to normalize a 16-bit value, returning a 16-bit result. Returns a shift count of 0 for an input of 0x0000.

NOTE

Does not actually normalize the value! This operation is *not* optimal on the DSP56800E because of the case of returning 0 for an input of 0x0000. See the intrinsic ffs_s which is more optimal but generates a different value for the case where the input == 0x0000.

Prototype

```
Word16 norm_s(Word16 ssrc)
```

Example

```
short s1 = 0x2000;/* .25 */
short result;


result = norm_s(s1);
// Expected value of result: 1
```

8.2.3.8.3 ffs_l

Computes the number of left shifts required to normalize a 32-bit value, returning a 16-bit result (finds 1st sign bit). Returns a shift count of 31 for an input of 0x00000000.

NOTE

Does not actually normalize the value! Also, see the intrinsic norm_l which handles the case where the input == 0x00000000 differently.

Prototype

```
Word16 ffs_l(Word32 lsrc)
```

Example

```
long ll = 0x20000000;/* .25 */
short result;


result = ffs_l(ll);
// Expected value of result: 1
```

8.2.3.8.4 norm_l

Computes the number of left shifts required to normalize a 32-bit value, returning a 16-bit result. Returns a shift count of 0 for an input of 0x00000000.

NOTE

Does not actually normalize the value! This operation is *not* optimal on the DSP56800E because of the case of returning 0 for an input of 0x00000000. See the intrinsic ffs_l which is more optimal but generates a different value for the case where the input == 0x00000000.

Prototype

```
Word16 norm_l(Word32 lsrc)
```

Example

```
long ll = 0x20000000;/* .25 */
short result;


result = norm_l(ll);
// Expected value of result: 1
```

8.2.3.9 Rounding

The intrinsic functions of the rounding group are:

- round_val
- ROUND_INT
- LL_ROUND

8.2.3.9.1 ROUND_INT

Rounds a 32-bit integer value into a 16-bit result. When an accumulator is the destination, zeroes out the LSP portion.

Prototype

```
Word16 __round_int(Word32 lvar1)
```

Example

```
long s = 0x12347FFF;
int result;
result = round_int (s);
// Expected value of result: 0x1234
```

8.2.3.9.2 round_val

Rounds a 32-bit fractional value into a 16-bit result. When an accumulator is the destination, zeroes out the LSP portion.

Assumptions

OMR's R bit was set to 1 at least three cycles before this code, that is, two's complement rounding, not convergent rounding.

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 round(Word32 lvar1)
```

Example

```
long l = 0x12348002;/*if low 16 bits = 0xFFFF > 0x8000 then add 1 */
short result;

result = round_val(l);
// Expected value of result: 0x1235
```

8.2.3.9.3 LL_ROUND

Rounds a 64-bit integer or fractional value. Returns a 32-bit value.

Prototype

```
Word32 __LL_round(Word64 llvar)
```

Example

```
long long s = 0x1234123443214321;
long result;
result = LL_round  (s);
// Expected value of result: 0x43214321
```

8.2.3.10 Shifting

The intrinsic functions of the shifting group are:

- shl
- shlftNs
- shlfts
- shr
- shr_r
- shrtNs
- L_shl
- L_shlftNs
- L_shlfts
- L_shr
- L_shr_r
- L_shrtNs

8.2.3.10.1 shl

Arithmetic shift of 16-bit value by a specified shift amount. If the shift count is positive, a left shift is performed. Otherwise, a right shift is performed. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

NOTE

This operation is not optimal on the DSP56800E because of the saturation requirements and the bidirectional capability. See the intrinsic shlftNs or shlfts which are more optimal.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 shl(Word16 sval2shft, Word16 s_shftamount)
```

Example

```
short result;

short s1 = 0x1234;

short s2 = 1;
```

```
result = shl(s1,s2);

// Expected value of result: 0x2468
```

8.2.3.10.2 shlftNs

Arithmetic shift of 16-bit value by a specified shift amount. If the shift count is positive, a left shift is performed. Otherwise, a right shift is performed. Saturation does not occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

NOTE

Ignores upper N-5 bits of `s_shftamount` except the sign bit (MSB). If `s_shftamount` is positive and the value in the lower 5 bits of `s_shftamount` is greater than 15, the result is 0. If `s_shftamount` is negative and the absolute value in the lower 5 bits of `s_shftamount` is greater than 15, the result is 0 if `sval2shft` is positive, and `0xFFFF` if `sval2shft` is negative.

Prototype

```
Word16 shlftNs(Word16 sval2shft, Word16 s_shftamount)
```

Example

```
short result;

short s1 = 0x1234;

short s2 = 1;


result = shlftNs(s1,s2);

// Expected value of result: 0x2468
```

8.2.3.10.3 shlfts

Arithmetic left shift of 16-bit value by a specified shift amount. Saturation does occur during a left shift if required. When an accumulator is the destination, zeroes out the LSP portion.

NOTE

This is not a bidirectional shift.

Assumptions

Assumed `s_shftamount` is positive.

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 shlfts(Word16 sval2shft, Word16 s_shftamount)
```

Example

```
short result;
short s1 = 0x1234;
short s2 = 3;

result = shlfts(s1,s2);
// Expected value of result: 0x91a0
```

8.2.3.10.4 shr

Arithmetic shift of 16-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

NOTE

This operation is not optimal on the DSP56800E because of the saturation requirements and the bidirectional capability. See the intrinsic shrtNs which is more optimal.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 shr(Word16 sval2shft, Word16 s_shftamount)
```

Example

```
short result;

short s1 = 0x2468;

short s2= 1;


result = shr(s1,s2);

// Expected value of result: 0x1234
```

8.2.3.10.5 shr_r

Arithmetic shift of 16-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. If a right shift is performed, then rounding performed on result. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

NOTE

This operation is not optimal on the DSP56800E because of the saturation requirements and the bidirectional capability. See the intrinsic shrtNs which is more optimal.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word16 shr_r(Word16 s_val2shft, Word16 s_shftamount)
```

Example

```
short result;

short s1 = 0x2468;

short s2= 1;


result = shr(s1,s2);

// Expected value of result: 0x1234
```

8.2.3.10.6 shrtNs

Arithmetic shift of 16-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. Saturation does not occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

NOTE

Ignores upper N-5 bits of `s_shftamount` except the sign bit (MSB). If `s_shftamount` is positive and the value in the lower 5 bits of `s_shftamount` is greater than 15, the result is 0 if `sval2shft` is positive, and `0xFFFF` if `sval2shft` is negative. If `s_shftamount` is negative and the absolute value in the lower 5 bits of `s_shftamount` is greater than 15, the result is 0.

Prototype

```
Word16 shrtNs(Word16 sval2shft, Word16 s_shftamount)
```

Example

```
short result;

short s1 = 0x2468;

short s2= 1;


result = shrtNs(s1,s2);

// Expected value of result: 0x1234
```

8.2.3.10.7 L_shl

Arithmetic shift of 32-bit value by a specified shift amount. If the shift count is positive, a left shift is performed. Otherwise, a right shift is performed. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

NOTE

This operation is not optimal on the DSP56800E because of the saturation requirements and the bidirectional capability. See the intrinsic L_shlftNs or L_shlfts which are more optimal.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_shl(Word32 lval2shft, Word16 s_shftamount)
```

Example

```
long result, l = 0x12345678;
short s2 = 1;

result = L_shl(l,s2);
// Expected value of result: 0x2468ACF0
```

8.2.3.10.8 L_shlftNs

Arithmetic shift of 32-bit value by a specified shift amount. If the shift count is positive, a left shift is performed. Otherwise, a right shift is performed. Saturation does not occur during a left shift.

NOTE

Ignores upper N-5 bits of s_shftamount except the sign bit (MSB).

Prototype

```
Word32 L_shlftNs(Word32 lval2shft, Word16 s_shftamount)
```

Example

```
long result, l = 0x12345678;
short s2= 1;

result = L_shlftNs(l,s2);
// Expected value of result: 0x2468ACF0
```

8.2.3.10.9 L_shlfts

Arithmetic left shift of 32-bit value by a specified shift amount. Saturation does occur during a left shift if required.

NOTE

This is not a bidirectional shift.

Assumptions

Assumed `s_shftamount` is positive.

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_shlfts(Word32 lval2shft, Word16 s_shftamount)
```

Example

```
long result, l = 0x12345678;
short s1 = 3;

result = shlfts(l, s1);
// Expected value of result: 0x91A259E0
```

8.2.3.10.10 L_shr

Arithmetic shift of 32-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. Saturation may occur during a left shift. When an accumulator is the destination, zeroes out the LSP portion.

NOTE

This operation is not optimal on the DSP56800E because of the saturation requirements and the bidirectional capability. See the intrinsic L_shrtNs which is more optimal.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_shr(Word32 lval2shft, Word16 s_shftamount)
```

Example

```
long result, l = 0x24680000;
short s2= 1;

result = L_shrtNs(l,s2);
// Expected value of result: 0x12340000
```

8.2.3.10.11 L_shr_r

Arithmetic shift of 32-bit value by a specified shift amount. If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. If a right shift is performed, then rounding performed on result. Saturation may occur during a left shift.

Assumptions

OMR's SA bit was set to 1 at least three cycles before this code, that is, saturation on data ALU results enabled.

Prototype

```
Word32 L_shr_r(Word32 lval2shft, Word16 s_shftamount)
```

Example

```
long l1 = 0x41111111;
short s2 = 1;
long result;

result = L_shr_r(l1,s2);
// Expected value of result: 0x20888889
```

8.2.3.10.12 L_shrtNs

Arithmetic shift of 32-bit value by a specified shift amount.If the shift count is positive, a right shift is performed. Otherwise, a left shift is performed. Saturation does not occur during a left shift.

NOTE

Ignores upper N-5 bits of `s_shftamount` except the sign bit (MSB).

Prototype

```
Word32 L_shrtNs(Word32 lval2shft, Word16 s_shftamount)
```

Example

```
long result, l = 0x24680000;

short s2= 1;


result = L_shrtNs(l,s2);

// Expected value of result: 0x12340000
```

8.2.4 Modulo Addressing Intrinsic Functions

A modulo buffer is a buffer in which the data pointer loops back to the beginning of the buffer once the pointer address value exceeds a specified limit.

The following figure depicts a modulo buffer with the limit six. Increasing the pointer address value to 0x106 makes it point to the same data it would point to if its address value were 0x100.

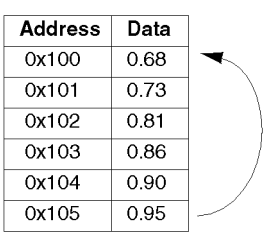

Figure 8-1. Example of Modulo Buffer

The CodeWarrior C compiler for DSP56800E uses intrinsic functions to create and manipulate modulo buffers. Normally, a modulo operation, such as the % operator, requires a runtime function call to the arithmetic library. For normally timed critical DSP loops, this binary operation imposes a large execution-time overhead.

The CodeWarrior implementation, however, replaces the runtime call with an efficient implementation of circular-address modification, either by using hardware resources or by manipulating the address mathematically.

Processors such as the DSP56800E have on-chip hardware support for modulo buffers. Modulo control registers work with the DSP pointer update addressing modes to access a range of addresses instead of a continuous, linear address space. But hardware support imposes strict requirements on buffer address alignment, pointer register resources, and limited modulo addressing instructions. For example, R0 and R1 are the only registers available for modulo buffers.

Accordingly, the CodeWarrior C compiler uses a well-defined set of intrinsic APIs to implement modulo buffers.

8.2.4.1 Modulo Addressing Intrinsic Functions

The intrinsic functions for modulo addressing are:

- __mod_init
- __mod_initint16
- __mod_start
- __mod_access
- __mod_update
- __mod_stop
- __mod_getint16
- __mod_setint16
- __mod_error

8.2.4.1.1 __mod_init

Initialize a modulo buffer pointer with arbitrary data using the address specified by the <addr_expr>. This function expects a byte address. <addr_expr> is an arbitrary C expression which normally evaluates the address at the beginning of the modulo buffer, although it may be any legal buffer address. The <mod_desc> evaluates to a compile time constant of either 0 or 1, represented by the modulo pointers R0 or R1, respectively. The <mod_sz> is a

compile time integer constant representing the size of the modulo buffer in bytes. The <data_sz> is a compile time integer constant representing the size of data being stored in the buffer in bytes. <data_sz> is usually derived from the sizeof() operator.

The __mod_init function may be called independently for each modulo pointer register.

If __mod_error has not been previously called, no record of __mod_init errors are saved.

If __mod_error has been previously called, __mod_init may set one of the error condition in the static memory location defined by __mod_error. (See __mod_error description for a complete list of error conditions).

Prototype

```
void __mod_init (
int <mod_desc>,
void * <addr_expr>,
int <mod_sz>,
int <data_sz>  );
```

Example

Initialize a modulo buffer pointer with a buffer size of 3 and where each element is a structure:

```
__mod_init(0, (void *)&struct_buf[0], 3, sizeof(struct
mystruct) );
```

8.2.4.1.2 __mod_initint16

Initialize modulo buffer pointer with integer data. The __mod_initint16 function behaves similarly to the __mod_init function, except that word addresses are used to initialize the modulo pointer register.

Prototype

```
void __mod_initint16(
int <mod_desc>,
int * <addr_expr>,
int <mod_sz> );
```

Example

Initialize an integer modulo buffer pointer with a buffer size of 10.

```
__mod_initint16(0, &int_buf[9], 10);
```

8.2.4.1.3 __mod_start

Write the modulo control register. The `__mod_start` function simply writes the modulo control register (M01) for each modulo pointer register which has been previously initialized. The values written to M01 depends on the size of the modulo buffer and which pointers have been initialized.

Prototype

```
void __mod_start( void )
```

8.2.4.1.4 __mod_access

Retrieve the modulo pointer. The `__mod_access` function returns the modulo pointer value specified by `<mod_desc>` in the R2 register, as per calling conventions. The value returned is a byte address. The data in the modulo buffer may be read or written by a cast and dereference of the resulting pointer.

Prototype

```
void *__mod_access( int <mod_desc>);
```

Example

Assign a value to the modulo buffer at the current pointer.

```
*((char *)__mod_access(0)) = (char)i;
```

8.2.4.1.5 __mod_update

Update the modulo pointer. The `__mod_update` function updates the modulo pointer by the number of data type units specified in `<amount>`. `<amount>` may be negative. Of course, the pointer will wrap to the beginning of the modulo buffer if the pointer is advanced beyond the modulo boundaries. `<amount>` must be a compile time constant.

Prototype

```
void  __mod_update( int <mod_desc>, int <amount>);
```

Example

Advance the modulo pointer by 2 units.

```
__mod_update(0, 2);
```

8.2.4.1.6 __mod_stop

Reset modulo addressing to linear addressing. This function writes the modulo control register with a value which restore linear addressing to the R0 and R1 pointer registers.

Prototype

```
void  __mod_stop( int <mod_desc );
```

8.2.4.1.7 __mod_getint16

Retrieve a 16-bit signed value from the modulo buffer and update the modulo pointer.This function returns an integer value from the location pointed to by the modulo pointer. The function then updates the modulo pointer by `<amount>` integer units (`<amount>*2 bytes`). `<amount>` must be a compile time constant.

Prototype

```
int __mod_getint16( int <mod_desc>, int <amount> );
```

Example

Retrieve an integer value from a modulo buffer and update the modulo buffer pointer by one word.

```
int y;
y = __mod_getint16(0, 1);
```

8.2.4.1.8 __mod_setint16

Write a 16-bit signed integer to the modulo buffer and update the pointer. This function evaluates <int_expr> and copies the value to the location pointed to by the modulo pointer. The modulo pointer is then updated by <amount>. <amount> must be a compile-time constant.

Prototype

```
int __mod_setint16( int <mod_desc>, int <int_expr>, int
<amount> );
```

Example

Write the modulo buffer with a value derived from an expression, do not update modulo pointer.

```
__mod_setint16( 0, getrandomint(), 0 );
```

8.2.4.1.9 __mod_error

Set up a modulo error variable. This function registers a static integer address to hold the error results from any of the modulo buffer API calls. The function returns 0 if it is successful, 1 otherwise. The argument must be the address of a static, global integer variable. This variable holds the result of calling each of the previously defined API functions. This allows the user to monitor the status of the error variable and take action if the error variable is non-zero. Typically, use __mod_error during development and remove it once debugging is complete. __mod_error generates no code, although the error variable may occupy a word of memory. A non-zero value in the error variable indicates a misuse of the one of the API functions. Once the error variable is set it is reset when __mod_stop is called. The error variable contains the error number of the last error. A successful call to an API function does not reset the error variable; only __mod_stop resets the error variable.

Prototype

```
int __mod_error( int * <static_object_addr>);
```

Example

Register the error number variable

```
static int myerrno;

assert( __mod_error(&myerrno) == 0 ) ;
```

8.2.4.2 Modulo Buffer Examples

The following listing is a modulo buffer example.

Listing: Modulo Buffer Example 1

```
#pragma define_section DATA_INT_MODULO ".data_int_modulo"



/* Place the buffer object in a unique section so the it can be aligned properly in the
linker control file. */


#pragma section DATA_INT_MODULO begin

int int_buf[10];

#pragma section DATA_INT_MODULO end



          /* Convenient defines for modulo descriptors */


#define M0 0

#define M1 1


int main ( void )

{

          int i;


/*   Modulo buffer will be initialized. R0 will be the modulo pointer register. The buffer
size is 10 units. The unit size is 'sizeof(int)'. */
```

```
  __mod_init(M0, (void *)&int_buf[0], 10, sizeof(int));


/* Write the modulo control register */


  __mod_start();


/* Write int_buf[0] through int_buf[9]. R0 initially points at int_buf[0] and wraps when the
pointer value exceeds int_buf[9]. The pointer is updated by 1 unit each time through the
loop */


          for ( i=0; i<100; i++ )
          {


          *((int *)__mod_access(M0)) = i;
          __mod_update(M0, 1);


          }


/* Reset modulo control register to linear addressing mode */
          __mod_stop();


}
```

The following listing is an another modulo buffer example.

Listing: Modulo Buffer Example 2

```
/* Set up a static location to save error codes */
if ( ! __mod_error(&err_codes)) {


printf ("__mod_error set up failed\n");
}


/* Initialize a modulo buffer pointer, pointing to an array of 10 ints. */


__mod_initint16(M0, &int_buf[9], 10);
```

```
/* Check for success of previous call */

          if ( err_code ) { printf ( "__mod_initint16 failed\n" ) };

          __mod_start();


/* Write modulo buffer with the result of the expression "i".
Decrement the buffer pointer for each execution of the loop.
The modulo buffer wraps from index 0 to 9 through the entire execution of the loop. */

          for ( i=100; i>0; i-- ) {

          __mod_setint16(M0, i, -1);

          }
          __mod_stop();
```

8.2.4.3 Points to Remember

As you use modulo buffer intrinsic functions, keep these points in mind:

- You must align modulo buffers properly, per the constraints that the *M56800E User's Manual* explains. There is no run-time validation of alignment. Using the modulo buffer API on unaligned buffers will cause erratic, unpredictable behavior during data accesses.
- Calling `__mod_start()` to write to the modulo control register effectively changes the hardware's global-address-generation state. This change of state affects all user function calls, run-time supporting function calls, standard library calls, and interrupts.
- You must account for any side-effects of enabling modulo addressing. Such a side-effect is that R0 and R1update in a modulo way.
- If you need just one modulo pointer is required, use the R0 address register. Enabling the R1 address register for modulo use also enables the R0 address register for modulo use. This is true even if `__mod_init()` or `__mod_initint16()` have not explicitly initialized R0.
- A successful API call does not clear the error code from the error variable. Only function `__mod_stop` clears the error code.

8.2.4.4 Modulo Addressing Error Codes

To register a static variable for error-code storage, use `__mod_error()`. If an error occurs, this static variable will contain one of the values the following table explains.

Table 8-4. Modulo Addressing Error Codes

Code	Meaning
11	<mod_desc> parameter must be zero or one.
12	R0 modulo pointer is already initialized. An extraneous call to `__mod_init` or `__mod_initint16` to initialize R0 has been made.
13	R1 modulo pointer is already initialized. An extraneous call to `__mod_init` or `__mod_initint16` to initialize R1 has been made.
14	Modulo buffer size must be a compile time constant.
15	Modulo buffer size must be greater than one.
16	Modulo buffer size is too big.
17	Modulo buffer size for R0 and R1 must be the same.
18	Modulo buffer data types for R0 and R1 must be the same.
19	Modulo buffer has not been initialized.
20	Modulo buffer has not been started.
21	Parameter is not a compile time constant.
22	Attempt to use word pointer functions with byte pointer initialization. `__mod_getint16` and `__mod_setint16` were called but `__mod_init` was used for initialization. `__mod_initint16` is required for pointer initialization.
23	Modulo increment value exceeds modulo buffer size.
24	Attempted use of R1 as a modulo pointer without initializing R0 for modulo use.

The following table lists the error codes possible for each modulo addressing intrinsic function.

Table 8-5. Possible Error Codes

Function	Possible Error Code
`__mod_init`	11, 12, 13, 14, 15, 16, 17, 18, 21
`__mod_stop`	none
`__mod_getint16`	11, 14, 20, 22, 24
`__mod_setint16`	11, 14, 20, 22, 24
`__mod_start`	none
`__mod_access`	11, 19, 20, 24

Table continues on the next page...

Table 8-5. Possible Error Codes (continued)

Function	Possible Error Code
__mod_update	11, 14, 20, 23, 24
__mod_initint16	11, 12, 13, 14, 15, 1 6, 17

Chapter 9
Pragmas

The #pragma preprocessor directive specifies option settings to the compiler.

This chapter describes how to use pragmas and lists the pragmas that the compiler recognizes:

9.1 Using Pragmas

Pragma settings may be manipulated to control the compiler's code generation. The compiler has additional capabilities to manage pragma settings themselves:

9.1.1 Checking Pragma Settings

The preprocessor function `__option()` returns the state of pragma settings at compile-time. The syntax is

```
__option(setting-name)
```

where *setting-name* is the name of a pragma that accepts the `on`, `off`, and `reset` options.

If *setting-name* is `on`, `__option(`*setting-name*`)` returns 1. If *setting-name* is `off`, `__option(`*setting-name*`)` returns 0. If *setting-name* is not the name of a pragma, `__option(`*setting-name*`)` returns false. If *setting-name* is the name of a pragma that does not accept the `on`, `off`, and `reset` options, the compiler issues a warning message.

The following listing shows an example.

Listing: Using the __option() Preprocessor Function

```
#if __option(ANSI_strict)
#include "portable.h" /* Use the portable declarations. */
#else
#include "custome.h" /* Use the specialized declarations. */
#endif
```

Table 9-1. Preprocessor Setting Names for __option()

Argument	Corresponding Setting or Pragma
always_inline	Pragma always_inline.
ANSI_strict	**ANSI Strict** setting in the **Language panel** and pragma ANSI_strict.
auto_inline	**Auto-Inline** setting of the **Inlining** menu in the Language panel and pragma auto_inline.
`check_inline_sp_effects`	Pragma `check_inline_sp_effects`.
const_strings	Pragma const_strings.
defer_codegen	Pragma defer_codegen.
dollar_identifiers	Pragma dollar_identifiers.
dont_inline	**Don't Inline** setting in the Language panel and pragma dont_inline.
dont_reuse_strings	**Reuse Strings** setting in the Language panel and pragma dont_reuse_strings.
`enumsalwaysint`	**Enums Always Int** setting in the Language panel and pragma `enumsalwaysint`.
`explicit_zero_data`	Pragma `explicit_zero_data`.
`factor1`	Pragma `factor1`.
`factor2`	Pragma `factor2`.
`factor3`	Pragma `factor3`.

Table continues on the next page...

Table 9-1. Preprocessor Setting Names for __option() (continued)

Argument	Corresponding Setting or Pragma
extended_errorcheck	**Extended Error Checking** setting in the **Language** panel and pragma extended_errorcheck.
fullpath_prepdump	Pragma fullpath_prepdump.
initializedzerodata	Pragma initializedzerodata.
inline_bottom_up	Pragma inline_bottom_up.
interrupt	Pragma interrupt.
line_prepdump	Pragma line_prepdump.
mpwc_newline	**Map newlines to CR** setting in the **Language** panel and pragma mpwc_newline.
mpwc_relax	**Relaxed Pointer Type Rules** setting in the **Language** panel and pragma mpwc_relax.
nofactor1	Pragma nofactor1.
nofactor2	Pragma nofactor2.
nofactor3	Pragma nofactor3.
only_std_keywords	**ANSI Keywords Only** setting in the **Language** panel and pragma only_std_keywords.
opt_common_subs	Pragma opt_common_subs.
opt_dead_assignments	Pragma opt_dead_assignments.
opt_dead_code	Pragma opt_dead_code.
opt_lifetimes	Pragma opt_lifetimes.
opt_loop_invariants	Pragma opt_loop_invariants.
opt_propagation	Pragma opt_propagation.
opt_strength_reduction	Pragma opt_strength_reduction.
opt_strength_reduction_strict	Pragma opt_strength_reduction_strict.
opt_unroll_loops	Pragma opt_unroll_loops.
optimize_for_size	Pragma optimize_for_size.
packstruct	Pragma pactstruct.
peephole	Pragma peephole.
pool_strings	**Pool Strings** setting in the **Language** panel and pragma pool_strings.
profile	Pragma profile.
readonly_strings	**Make String Read Only** setting in the M56800 Processor settings panel and pragma readonly_strings.
require_prototypes	**Require Function Prototypes** setting in the Language panel and pragma require_prototypes.
reverse_bitfields	Pragma reverse_bitfields.
simple_prepdump	Pragma simple_prepdump.
suppress_init_code	Pragma suppress_init_code.
suppress_warnings	Pragma suppress_warnings.
syspath_once	Pragma syspath_once.

Table continues on the next page...

Table 9-1. Preprocessor Setting Names for __option() (continued)

Argument	Corresponding Setting or Pragma
unsigned_char	**Use Unsigned Chars** setting in the **Language** panel and pragma unsigned_char.
warn_any_ptr_int_conv	**Pragma**warn_any_ptr_int_conv.
warn_emptydecl	**Empty Declarations** setting in the Language panel and pragma warn_emptydecl.
warn_extracomma	**Extra Commas** setting in the Preprocessor panel and pragma warn_extracomma.
warn_filenamecaps	Pragma warn_filenamecaps.
warn_filenamecaps_system	Pragma warn_filenamecaps_system.
warn_illegal_instructions	Pragma warn_illegal_instructions.
warn_illpragma	**Illegal Pragmas** setting in the panel and pragma warn_illpragma.
warn_impl_f2i_conv	Pragma warn_impl_f2i_conv.
warn_impl_i2f_conv	Pragma warn_impl_i2f_conv.
warn_impl_s2u_conv	Pragma warn_impl_s2u_conv.
warn_implicitconv	**Implicit Arithmetic Conversions** setting in the processor panel and pragma warn_implicitconv.
warn_largeargs	Pragma warn_largeargs.
warn_missingreturn	Pragma warn_missingreturn
warn_no_side_effect	Pragma warn_no_side_effect.
warn_notinlined	**Non-Inlined Functions** setting in the processor panel and pragma warn_notinlined.
warn_padding	Pragma warn_padding.
warn_possunwant	**Possible Errors** setting in the Preprocessor panel and pragma warn_possunwant.
warn_ptr_int_conv	Pragma warn_ptr_int_conv
warn_resultnotused	Pragma warn_resultnotused.
warn_undefmacro	Pragma warn_undefmacro.
warn_unusedarg	**Unused Arguments** setting in the processor panel and pragma warn_unusedarg.
warn_unusedvar	**Unused Variables** setting in the Language panel and pragma warn_unusedvar.
warning_errors	**Treat Warnings As Errors** setting in the Preprocessor panel and pragma warning_errors.

9.1.2 Saving and Restoring Pragma Settings

There are some occasions when you would like to apply pragma settings to a piece of source code independently from the settings in the rest of the source file. For example, a function might require unique optimization settings that should not be used in the rest of the function's source file.

Remembering which pragmas to save and restore is tedious and error-prone. Fortunately, the compiler has mechanisms that save and restore pragma settings at compile time. All pragma settings and some individual pragma settings may be saved at one point in a compilation unit (a source code file and the files that it includes), changed, then restored later in the same compilation unit. Pragma settings cannot be saved in one source code file then restored in another unless both source code files are included in the same compilation unit.

Pragmas `push` and `pop` save and restore, respectively, most pragma settings in a compilation unit. Pragmas `push` and `pop` may be nested to unlimited depth. The following listing shows an example.

Listing: Using push and pop to Save and Restore Pragma Settings

```
/* Settings for this file. */
#pragma opt_unroll_loops on

#pragma optimize_for_size off

void fast_func_A(void)

{

/* ... */

}

/* Settings for slow_func(). */

#pragma push /* Save file settings. */

#pragma optimization_size 0

void slow_func(void)

{

/* ... */

}

#pragma pop /* Restore file settings. */

void fast_func_B(void)

{

/* ... */

}
```

Pragmas that have a `reset` option perform the same actions as pragmas `push` and `pop`, but apply to a single pragma. A pragma's `on` and `off` settings save the pragma's current setting before changing it to the new setting. A pragma's `reset` option restores the pragma's setting. The `on`/ `off` and `reset` options may be nested to an unlimited depth. The following listing shows an example.

Listing: Using the Reset Option to Save and Restore a Pragma Setting

```
/* Setting for this file. */
#pragma opt_unroll_loops on

void fast_func_A(void)

{

/* ... */

}

/* Setting for smallslowfunc(). */

#pragma opt_unroll_loops off

void small_func(void)

{

/* ... */

}

/* Restore previous setting. */

#pragma opt_unroll_loops reset

void fast_func_B(void)

{

/* ... */

}
```

9.1.3 Determining which Settings are Saved and Restored

Not all pragma settings are saved and restored by pragmas `push` and `pop`. Pragmas that do not change compiler settings are not affected by `push` and `pop`. For example, pragma `message` cannot be saved and restored.

The following listing shows an example that checks if the `ANSI_strict` pragma setting is saved and restored by pragmas `push` and `pop`.

Listing: Testing if Pragma's Push and Pop Save and Restore a Setting

```
/* Preprocess this source code. */
#pragma ANSI_strict on

#pragma push

#pragma ANSI_strict off

#pragma pop

#if __option(ANSI_strict)

#error "Saved and restored by push and pop."

#else

#error "Not affected by push and pop."
```

```
#endif
```

9.1.4 Illegal Pragmas

If you enable the **Illegal Pragmas** setting, the compiler issues a warning when it encounters a pragma it does not recognize. For example, the pragma statements in The following listing generate warnings with the **Illegal Pragmas** setting enabled.

Listing: Illegal Pragmas

```
#pragma near_data off       // WARNING: near_data is not a pragma.
#pragma ANSI_strict select  // WARNING: select is not defined


#pragma ANSI_strict on      // OK
```

The **Illegal Pragmas** setting corresponds to the pragma `warn_illpragma`, described at warn_illpragma. To check this setting, use `__option (warn_illpragma)`.

See Checking Pragma Settings for information on how to use this directive.

9.2 Pragma Scope

The scope of a pragma setting is limited to a compilation unit (a source code file and the files that it includes).

At the beginning of compilation unit, the compiler uses its default settings. The compiler then uses the settings specified by the CodeWarrior IDE's build target or in command-line options.

The compiler uses the setting in a pragma beginning at the pragma's location in the compilation unit. The compilers continues using this setting:

- Until another instance of the same pragma appears later in the source code
- Until an instance of pragma `pop` appears later in the source code
- Until the compiler finishes translating the compilation unit

9.3 Standard C and C++ Conformance Pragmas

The 56800x has the following pragmas:

- ANSI_strict
- only_std_keywords

9.3.1 ANSI_strict

Controls the use of non-standard language features.

Syntax

```
#pragma ANSI_strict on | off | reset
```

Remarks

If you enable the pragma `ANSI_strict`, the compiler generates an error if it encounters any of the following common ANSI extensions:

- C++-style comments. The following listing shows an example.
  Listing: C++ Comments

  ```
  a = b;    // This is a C++-style comment
  ```

- Unnamed arguments in function definitions. The following listing shows an example.
  Listing: Unnamed Arguments

  ```
  void f(int ) {} /* OK, if ANSI Strict is disabled */

  void f(int i) {} /* ALWAYS OK                        */
  ```

- A `#` token that does not appear before an argument in a macro definition. The following listing shows an example.
  Listing: Using # in Macro Definitions

  ```
  #define add1(x) #x #1

        /* OK, if ANSI_strict is disabled,


           but probably not what you wanted:


           add1(abc) creates "abc"#1            */


  #define add2(x) #x "2"


        /* ALWAYS OK: add2(abc) creates "abc2"  */
  ```

- An identifier after `#endif`. The following listing shows an example.

Listing: Identifiers After #endif

```
#ifdef __MWERKS__
  /* . . . */

#endif __MWERKS__       /* OK, if ANSI_strict is disabled */

#ifdef __MWERKS__

  /* . . . */

#endif /*__MWERKS__*/   /* ALWAYS OK  */
```

This pragma corresponds to the **ANSI Strict** setting in the Language panel. To check this setting, use `__option (ANSI_strict)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.3.2 only_std_keywords

Controls the use of ISO keywords.

Syntax

```
#pragma only_std_keywords on | off | reset
```

Remarks

The C/C++ compiler recognizes additional reserved keywords. If you are writing code that must follow the ANSI standard strictly, enable the pragma `only_std_keywords`.

This pragma corresponds to the **ANSI Keywords Only** setting in the Language panel. To check this setting, use `__option (only_std_keywords)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.4 Language Translation and Extensions Pragmas

The 56800x has the following pragmas:

- gcc_extensions

- mpwc_newline
- mpwc_relax

9.4.1 gcc_extensions

Controls the acceptance of GNU C language extensions.

Syntax

```
#pragma gcc_extensions on | off | reset
```

Remarks

If you enable this pragma, the compiler accepts GNU C extensions in C source code. This includes the following non-ANSI C extensions:

- Initialization of automatic `struct` or `array` variables with non- `const` values. The following listing provides an example.
  Listing: Example of Array Initialization with a Non-const Value

```
int foo(int arg)
{

    int arr[2] = { arg, arg+1 };

}
```

- `sizeof(void)==1`
- `sizeof(`*function-type*`)==1`
- Limited support for GCC statements and declarations within expressions. The following listing provides an example.
  Listing: Example of GCC Statements and Declarations Within Expressions

```
#pragma gcc_extensions on
#define POW2(n) ({ int i,r; for(r=1,i=n; i>0; --i) r<<=1; r;})

int main()

{

    return POW2(4);
```

```
}
```

This feature only works for expressions in function bodies.

- Macro redefinitions without a previous `#undef`.
- The GCC keyword `typeof`.

This pragma does not correspond to any setting in the Language panel. To check the global optimizer, use `__option (gcc_extensions)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.4.2 mpwc_newline

Controls the use of newline character convention used by the Apple MPW C.

Syntax

```
#pragma mpwc_newline on | off | reset
```

Remarks

If you enable this pragma, the compiler uses the MPW conventions for the `'\n'` and `'\r'` characters. Otherwise, the compiler uses the Freescale C/C++ conventions for these characters.

In MPW, '\ n' is a Carriage Return (0x0D) and `'\r'` is a Line Feed (0x0A). In Freescale C/C++, they are reversed: '\n' is a Line Feed and `'\r'` is a Carriage Return.

If you enable this pragma, use ANSI C/C++ libraries that were compiled when this pragma was enabled.

If you enable this pragma and use the standard ANSI C/C++ libraries, you cannot read and write `'\n'` and `'\r'` properly. For example, printing `'\n'` brings you to the beginning of the current line instead of inserting a newline.

This pragma corresponds to the **Map newlines to CR** setting in the Language panel. To check this setting, use `__option (mpwc_newline)`, described in Checking Pragma Settings. By default, this pragma is disabled.

Enabling this setting is not useful for the DSP56800 target.

9.4.3 mpwc_relax

Controls the compatibility of the `char*` and `unsigned char*` types.

Syntax

```
#pragma mpwc_relax on | off | reset
```

Remarks

If you enable this pragma, the compiler treats `char*` and `unsigned char*` as the same type. This setting is especially useful if you are using code written before the ANSI C standard. This old source code frequently used these types interchangeably. This setting has no effect on C++ source code.

You can use this pragma to relax function pointer checking:

```
#pragma mpwc_relax on

extern void f(char *);

extern void(*fp1)(void *) = &f;          // error but allowed

extern void(*fp2)(unsigned char *) = &f; // error but allowed
```

This pragma corresponds to the **Relaxed Pointer Type Rules** setting in the Language panel. To check this setting, `__option (mpwc_relax)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.5 Errors, Warnings, and Diagnostic Control Pragmas

The 56800x has the following pragmas:

- check_c_src_pipeline
- check_inline_asm_pipeline
- check_inline_sp_effects
- extended_errorcheck
- require_prototypes
- suppress_init_code
- suppress_warnings
- unsigned_char
- unused
- warn_any_ptr_int_conv
- warn_emptydecl
- warn_extracomma
- warn_filenamecaps

- warn_filenamecaps_system
- warn_illpragma
- warn_impl_f2i_conv
- warn_impl_i2f_conv
- warn_impl_s2u_conv
- warn_implicitconv
- warn_largeargs
- warn_missingreturn
- warn_no_side_effect
- warn_notinlined
- warn_padding
- warn_possiblyuninitializedvar
- warn_possunwant
- warn_ptr_int_conv
- warn_resultnotused
- warn_undefmacro
- warn_uninitializedvar
- warn_unusedarg
- warn_unusedvar
- warning_errors

9.5.1 check_c_src_pipeline

This pragma controls detection of a pipeline conflict in the C language code.

Compatibility

This pragma is not compatible with the DSP56800 compiler, but it is compatible with the DSP56800E compiler.

Syntax

```
#pragma check_c_src_pipeline [off|conflict]
```

Remarks

Use this pragma for extra validation of generated C code. The compiler already checks for pipeline conflicts; this pragma tells the compiler to add another check for pipeline conflicts. Should this pragma detect a pipeline conflict, it issues an error message.

NOTE

The pipeline conflicts that this pragma finds are rare. Should this pragma report such a conflict with your code, you should report the matter to Freescale.

9.5.2 check_inline_asm_pipeline

This pragma controls detection of a pipeline conflicts and stalls in assembly language source code.

Compatibility

This pragma is not compatible with the DSP56800 compiler, but it is compatible with the DSP56800E compiler.

Syntax

```
#pragma check_inline_asm_pipeline
[off|conflict|conflict_and_stall]
```

Remarks

Use this pragma to detect a source-code, assembly language pipeline conflict or stall, then generate an error message. In some cases, the source code can be a mix of assembly language and C language.

The option `conflict` only detects and generates error messages for pipeline conflict.

The option `conflict_and_stall` detects and generates error messages for pipeline conflicts and stalls.

9.5.3 check_inline_sp_effects

Generates a warning if the user specifies an inline assembly instruction which modifies the SP by a run-time dependent amount.

Syntax

```
#pragma check_inline_sp_effects on | off | reset
```

Remarks

If this pragma is not specified off, instructions which modify the SP by a run-time dependent amount are ignored. In this case, stack-based references may be silently wrong. This pragma is added for compatibility with existing code which may have run-time modifications of the SP already. However, known compile times inconsistencies in SP modifications are always flagged as errors, since the SP must be correct to return from functions.

This pragma does not correspond to any panel setting in the Warnings panel. To check this setting, use `__option (check_inline_sp_effects)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.5.4 extended_errorcheck

Controls the issuing of warnings for possible unintended logical errors.

Syntax

```
#pragma extended_errorcheck on | off | reset
```

Remarks

If you enable this pragma, the C compiler generates a warning (not an error) if it encounters some common programming errors.

This pragma corresponds to the **Extended Error Checking** setting in the Warnings panel. To check this setting, use `__option (extended_errorcheck)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.5.5 require_prototypes

Controls whether or not the compiler should expect function prototypes.

Syntax

```
#pragma require_prototypes on | off | reset
```

Remarks

This pragma only works for non-static functions.

If you enable this pragma, the compiler generates an error if you use a function that does not have a prototype. This pragma helps you prevent errors that happen when you use a function before you define it or refer to it.

This pragma corresponds to the **Require Function Prototypes** setting in the Language panel. To check this setting, use `__option (require_prototypes)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.5.6 suppress_init_code

Controls the suppression of static initialization object code.

Syntax

```
#pragma suppress_init_code on | off | reset
```

Remarks

If you enable this pragma, the compiler does not generate any code for static data initialization.

CAUTION

Beware when using this pragma because it can produce erratic or unpredictable behavior in your program.

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (suppress_init_code)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.5.7 suppress_warnings

Controls the issuing of warnings.

Syntax

```
#pragma suppress_warnings on | off | reset
```

Remarks

If you enable this pragma, the compiler does not generate warnings, including those that are enabled.

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (suppress_warnings)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.5.8 unsigned_char

Controls whether or not declarations of type `char` are treated as `unsigned char`.

Syntax

```
#pragma unsigned_char on | off | reset
```

Remarks

If you enable this pragma, the compiler treats a `char` declaration as if it were an `unsignedchar` declaration.

NOTE

If you enable this pragma, your code might not be compatible with libraries that were compiled when the pragma was disabled. In particular, your code might not work with the ANSI libraries included with CodeWarrior.

This pragma corresponds to the **Use Unsigned Chars** setting in the Language panel. To check this setting, use `__option(unsigned_char)`, described in Checking Pragma Settings. By default, this setting is disabled.

9.5.9 unused

Controls the suppression of warnings for variables and parameters that are not referenced in a function.

Syntax

```
#pragma unused ( var_name [, var_name ]... )
```

Remarks

This pragma suppresses the compile time warnings for the unused variables and parameters specified in its argument list. You can use this pragma only within a function body, and the listed variables must be within the scope of the function.

Listing: Example of Pragma unused() in C

```
#pragma warn_unusedvar on  // See pragma  warn_unusedvar.

#pragma warn_unusedarg on  // See pragma  warn_unusedarg.
```

```
static void ff(int a)

{

  int b;

#pragma unused(a,b)  // Compiler does not warn

                     // that a and b are unused

                     // . . .

}
```

This pragma does not correspond to any panel setting in the Language panel. By default, this pragma is disabled.

9.5.10 warn_any_ptr_int_conv

Controls if the compiler generates a warning when an integral type is explicitly converted to a pointer type or vice versa.

Syntax

```
#pragma warn_any_ptr_int_conv on | off | reset
```

Remarks

This pragma is useful to identify potential pointer portability issues. An example is shown in the following listing.

Listing: Example of warn_any_ptr_int_conv

```
#pragma warn_ptr_int_conv on
short i, *ip

void foo() {

   i = (short)ip;    // WARNING: integral type is not large
```

```
                    // large enough to hold pointer

}

#pragma warn_any_ptr_int_conv on

void bar() {

   i  = (int)ip;     // WARNING: pointer to integral

                     // conversion

   ip = (short *)i;  // WARNING: integral to pointer

                     // conversion

}
```

See also warn_ptr_int_conv.

This pragma corresponds to the **Pointer/Integral Conversions** setting in the Warnings panel. To check this setting, use `__option (`warn_any_ptr_int_conv `)`, described in Checking Pragma Settings. By default, this pragma is `off`.

9.5.11 warn_emptydecl

Controls the recognition of declarations without variables.

Syntax

```
#pragma warn_emptydecl on | off | reset
```

Remarks

If you enable this pragma, the compiler displays a warning when it encounters a declaration with no variables.

Listing: Example of Pragma warn_emptydecl

```
int ;     // WARNING
int i;    // OK
```

This pragma corresponds to the **Empty Declarations** setting in the Warnings panel. To check this setting, use `__option (warn_emptydecl)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.5.12 warn_extracomma

Controls the recognition of superfluous commas.

Syntax

```
#pragma warn_extracomma on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning when it encounters an extra comma.

Listing: Example of Pragma warn_extracomma

```
enum {l,m,n,o,};  // WARNING: When the warning is enabled, it will
                  // generate :
```

This pragma corresponds to the **Extra Commas** setting in the Warnings panel. To check this setting, use `__option (warn_extracomma)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.5.13 warn_filenamecaps

Controls the recognition of conflicts involving case-sensitive filenames within user includes.

Syntax

```
#pragma warn_filenamecaps on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning when an `include` directive capitalizes a filename within a user include differently from the way the filename appears on a disk. It also recognizes 8.3 DOS filenames in Windows when a long filename is available. This pragma helps avoid porting problems to operating systems with case-sensitive filenames.

By default, this pragma only checks the spelling of user includes such as the following:

```
#include "file"
```

For more information on checking system includes, see warn_filenamecaps_system.

This pragma does not correspond to any panel setting in the Warnings panel. To check this setting, use `__option (warn_filenamecaps)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.5.14 warn_filenamecaps_system

Controls the recognition of conflicts involving case-sensitive filenames within system includes.

Syntax

```
#pragma warn_filenamecaps_system on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning when an `include` directive capitalizes a filename within a system include differently from the way the filename appears on a disk. It also recognizes 8.3 DOS filenames in Windows when a long filename is available. This pragma helps avoid porting problems to operating systems with case-sensitive filenames.

To check the spelling of system includes such as the following:

```
#include <file>
```

use this pragma along with the warn_filenamecaps pragma.

This pragma does not correspond to any panel setting in the Warnings panel. To check this setting, use `__option (warn_filenamecaps_system)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.5.15 warn_illpragma

Controls the recognition of illegal pragma directives.

Syntax

```
#pragma warn_illpragma on | off | reset
```

Remarks

If you enable this pragma, the compiler displays a warning when it encounters a pragma it does not support. For more information about this warning, see Illegal Pragmas.

This pragma corresponds to the **Illegal Pragmas** setting in the Language panel. To check this setting, use `__option (warn_illpragma)`, described in Checking Pragma Settings. By default, this setting is disabled.

9.5.16 warn_impl_f2i_conv

Controls the issuing of warnings for implicit `float`-to- `int` conversions.

Syntax

```
#pragma warn_impl_f2i_conv on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning for implicitly converting floating-point values to integral values. The following listing provides an example.

Listing: Example of Implicit float-to-int Conversion

```
#pragma warn_implicit_conv on
#pragma warn_impl_f2i_conv on


float f;


signed int si;


int main()


{
```

```
    si = f;     // WARNING


#pragma warn_impl_f2i_conv off


    si = f;     // OK


}
```

Use this pragma along with the warn_implicitconv pragma.

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (warn_impl_f2i_conv)`, described in Checking Pragma Settings. By default, this pragma is enabled.

9.5.17 warn_impl_i2f_conv

Controls the issuing of warnings for implicit `int`-to- `float` conversions.

Syntax

```
#pragma warn_impl_i2f_conv on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning for implicitly converting integral values to floating-point values. The following listing provides an example.

Listing: Example of Implicit <codeph>int</codeph>-to-<codeph>float</codeph> Conversion

```
#pragma warn_implicit_conv on
#pragma warn_impl_i2f_conv on


float f;


signed int si;


int main()


{
```

```
    f = si;      // WARNING


#pragma warn_impl_i2f_conv off


    f = si;     // OK


}
```

Use this pragma along with the warn_implicitconv pragma.

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (warn_impl_i2f_conv)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.5.18 warn_impl_s2u_conv

Controls the issuing of warnings for implicit conversions between the `signed int` and `unsigned int` data types.

Syntax

```
#pragma warn_impl_s2u_conv on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning for implicitly converting either from `signed int` to `unsigned int` or vice versa. The following listing provides an example.

Listing: Example of Implicit Conversions Between Signed int and Unsigned int

```
#pragma warn_implicit_conv on
#pragma warn_impl_s2u_conv on


signed int si;


unsigned int ui;


int main()
```

```
{

    ui = si;    // WARNING

    si = ui;    // WARNING

#pragma warn_impl_s2u_conv off

    ui = si;    // OK

    si = ui;    // OK

}
```

Use this pragma along with the warn_implicitconv pragma.

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (warn_impl_s2u_conv)`, described in Checking Pragma Settings. By default, this pragma is enabled.

9.5.19 warn_implicitconv

Controls the issuing of warnings for all implicit arithmetic conversions.

Syntax

```
#pragma warn_implicitconv on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning for all implicit arithmetic conversions when the destination type might not represent the source value. The following listing provides an example.

Listing: Example of Implicit Conversion

```
#pragma warn_implicitconv on

float f;


signed int si;
```

```
unsigned int ui;


int main()


{


    f  = si;    // OK


    si = f;     // WARNING


    ui = si;    // WARNING


    si = ui;    // WARNING


}
```

The default setting for `warn_impl_i2fconf` pragma is disabled. Use the `warn_implicitconv` pragma along with the `warn_impl_i2f_conv` pragma to generate the warning for the int-to-float conversion.

This pragma corresponds to the **Implicit Arithmetic Conversions** setting in the Language panel. To check this setting, use `__option (warn_implicitconv)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.5.20 warn_largeargs

Controls the issuing of warnings for passing non-integer numeric values to undeclared functions.

Syntax

```
#pragma warn_largeargs on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning if you attempt to pass a non-integer numeric value, such as a `float` or `long long`, to an undeclared function when the require_prototypes pragma is disabled.

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (warn_largeargs)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.5.21 warn_missingreturn

Issues a warning when a function that returns a value is missing a `return` statement.

Syntax

```
#pragma warn_missingreturn on | off | reset
```

Remarks

An example is shown in the following listing.

Listing: Example of warn_missingreturn pragma

```
#pragma warn_missingreturn on
int foo()
{
            // no return statement in foo()
}           // generates a warning: return value expected
```

This pragma corresponds to the **Missing `return' Statements** option in the Language panel. To check this setting, use `__option(warn_missingreturn)`, described in Checking Pragma Settings.

By default, this pragma is set to the same value as `__option(extended_errorcheck)`.

9.5.22 warn_no_side_effect

Controls the issuing of warnings for redundant statements.

Syntax

```
#pragma warn_no_side_effect on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning when it encounters a statement that produces no side effect. To suppress this warning, cast the statement with `(void)`. The following listing provides an example.

Listing: Example of Pragma warn_no_side_effect

```
#pragma warn_no_side_effect on

void foo(int a,int b)


{


   a+b;          // WARNING: expression has no side effect


   (void)(a+b);  // void cast suppresses warning


}
```

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (warn_no_side_effect)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.5.23 warn_notinlined

Controls the issuing of warnings for functions the compiler cannot inline.

Syntax

```
#pragma warn_notinlined on | off | reset
```

Remarks

The compiler issues a warning for non-inlined inline function calls.

This pragma corresponds to the **Non-Inlined Functions** setting in the Language panel. To check this setting, use `__option (warn_notinlined)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.5.24 warn_padding

Controls the issuing of warnings for data structure padding.

Syntax

```
#pragma warn_padding on | off | reset
```

Remarks

If you enable this pragma, the compiler warns about any bytes that were implicitly added after an ANSI C `struct` member to improve memory alignment.

This pragma corresponds to the **Pad Bytes Added** setting in the Language panel. To check this setting, use `__option (warn_padding)`, described in Checking Pragma Settings. By default, this setting is disabled.

9.5.25 warn_possiblyuninitializedvar

This pragma is different from `warn_uninitializedvar.` that uses a slightly different process to detect the uninitialized variables. It gives a warning whenever local variables are used before being initialized along any path to the usage. As a result, you get more warnings. However, some of the warnings are false ones. The warnings will be false when all paths with uninitialized status turn out to be paths that can never actually be taken.

NOTE

`warn_possiblyuninitializedvar` is a superset of warn_uninitializedvar

Syntax

```
#pragma warn_possiblyuninitializedvar on | off | reset
```

Remarks

This pragma has no corresponding setting in the CodeWarrior IDE. By default, this pragma is off.

9.5.26 warn_possunwant

Controls the recognition of possible unintentional logical errors.

Syntax

```
#pragma warn_possunwant on | off | reset
```

Remarks

If you enable this pragma, the compiler checks for common errors that are legal C/C++ but might produce unexpected results, such as putting in unintended semicolons or confusing `=` and `==`.

This pragma corresponds to the **Possible Errors** setting in the Language panel. To check this setting, use `__option (warn_possunwant)`, described in Checking Pragma Settings. By default, this setting is disabled.

9.5.27 warn_ptr_int_conv

Controls the recognition the conversion of pointer values to incorrectly-sized integral values.

Syntax

```
#pragma warn_ptr_int_conv on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning if an expression attempts to convert a pointer value to an integral type that is not large enough to hold the pointer value.

Listing: Example for #pragma warn_ptr_int_conv

```
#pragma warn_ptr_int_conv on

char *my_ptr;


char too_small = (char)my_ptr;  // WARNING: char is too small
```

See also warn_any_ptr_int_conv.

This pragma corresponds to the **Pointer / Integral Conversions** setting in the Language panel. To check this setting, use `__option (warn_ptr_int_conv)`, described in Checking Pragma Settings. By default, this setting is disabled.

9.5.28 warn_resultnotused

Controls the issuing of warnings when function results are ignored.

Syntax

```
#pragma warn_resultnotused on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning when it encounters a statement that calls a function without using its result. To prevent this, cast the statement with `(void)`. The following listing provides an example.

Listing: Example of Function Calls with Unused Results

```
#pragma warn_resultnotused on

extern int bar();


void foo()


{


   bar();           // WARNING: result of function call is not used


   (void)bar();      // `void' cast suppresses warning


}
```

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (warn_resultnotused)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.5.29 warn_undefmacro

Controls the detection of undefined macros in `#if/#elif` conditionals.

Syntax

```
#pragma warn_undefmacro on | off | reset
```

Remarks

The following listing provides an example.

Listing: Example of Undefined Macro

```
#if UNDEFINEDMACRO == 4 // WARNING: undefined macro
                        // 'UNDEFINEDMACRO' used in
                        // #if/#elif conditional
```

Use this pragma to detect the use of undefined macros (especially expressions) where the default value 0 is used.

NOTE

A warning is only issued when a macro is evaluated. A short-circuited " `&&`" or " `||`" test or unevaluated " `?:`" will not produce a warning.

This pragma corresponds to the **Undefined Macro in #if** setting in the Language panel. To check this setting, use `__option (`warn_undefmacro `)`, described in Checking Pragma Settings. By default, this pragma is `off`.

9.5.30 warn_uninitializedvar

Controls the compiler to perform data flow analysis and emits a warning message whenever there is a usage of a local variable and no path exists from any initialization of the same local variable.

Usages will not receive a warning if the variable is initialized along any path to the usage, even though the variable may be uninitialized along some other path. The `warn_possiblyunitializedvar` pragma is introduced for such cases. Refer to pragma warn_possiblyuninitializedvar for more details.

Syntax

```
#pragma warn_uninitializedvar on | off | reset
```

Remarks

This pragma has no corresponding setting in the CodeWarrior IDE. By default, this pragma is on.

9.5.31 warn_unusedarg

Controls the recognition of unreferenced arguments.

Syntax

```
#pragma warn_unusedarg on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning when it encounters an argument you declare but do not use. To suppress this warning in C++ source code, leave an argument identifier out of the function parameter list.

This pragma corresponds to the **Unused Arguments** setting in the Language panel. To check this setting, use `__option (warn_unusedarg)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.5.32 warn_unusedvar

Controls the recognition of unreferenced variables.

Syntax

```
#pragma warn_unusedvar on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning when it encounters a variable you declare but do not use. To suppress this warning in C++ source code, leave an argument identifier out of the function parameter list.

This pragma corresponds to the **Unused Variables** setting in the Language panel. To check this setting, use `__option (warn_unusedvar)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.5.33 warning_errors

Controls whether or not warnings are treated as errors.

Syntax

```
#pragma warning_errors on | off | reset
```

Remarks

If you enable this pragma, the compiler treats all warnings as though they were errors and does not translate your file until you resolve them.

This pragma corresponds to the **Treat All Warnings as Errors** setting in the Language panel. To check this setting, use `__option (warning_errors)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.6 Preprocessing and Precompilation Pragmas

The 56800x has the following pragmas:

- dollar_identifiers
- fullpath_prepdump
- mark
- notonce
- once
- pop, push
- syspath_once

9.6.1 dollar_identifiers

Controls use of dollar signs ($) in identifiers.

Syntax

```
#pragma dollar_identifiers on | off | reset
```

Remarks

If you enable this pragma, the compiler accepts dollar signs ($) in identifiers. Otherwise, the compiler issues an error if it encounters anything but underscores, alphabetic, and numeric characters in an identifier.

This pragma does not correspond to any panel setting. To check this setting, use the `__option (dollar_identifiers)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.6.2 fullpath_prepdump

Shows the full path of included files in preprocessor output.

Syntax

```
#pragma fullpath_prepdump on | off | reset
```

Remarks

If you enable this pragma, the compiler shows the full paths of files specified by the `#include` directive as comments in the preprocessor output. Otherwise, only the file name portion of the path appears.

This pragma does not correspond to any panel setting. To check this setting, use the `__option (fullpath_prepdump)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.6.3 mark

Adds an item to the **Function** pop-up menu in the IDE editor.

Syntax

```
#pragma mark itemName
```

Remarks

This pragma adds *itemName* to the source file's **Function** pop-up menu. If you open the file in the CodeWarrior Editor and select the item from the **Function** pop-up menu, the editor brings you to the pragma. Note that if the pragma is inside a function definition, the item does not appear in the **Function** pop-up menu.

If *itemName* begins with "-", a menu separator appears in the IDE's **Function** pop-up menu:

```
#pragma mark -
```

This pragma does not correspond to any setting in the Language panel. By default, this pragma is disabled.

9.6.4 notonce

Controls whether or not the compiler lets included files be repeatedly included, even with `#pragma once on`.

Syntax

```
#pragma notonce
```

Remarks

If you enable this pragma, `include` statements can be repeatedly included, even if you have enabled `#pragmaonceon`. For more information, see once.

This pragma does not correspond to any setting in the Language panel. By default, this pragma is disabled.

9.6.5 once

Controls whether or not a header file can be included more than once in the same source file.

Syntax

```
#pragma once [ on ]
```

Remarks

Use this pragma to ensure that the compiler includes header files only once in a source file.

There are two versions of this pragma: `#pragma once` and `#pragma once on`. Use `#pragma once` in a header file to ensure that the header file is included only once in a source file. Use `#pragma once on` in a header file or source file to insure that *any* file is included only once in a source file.

This pragma does not correspond to any setting in the Language panel. By default, this pragma is disabled.

9.6.6 pop, push

Save and restore pragma settings.

Syntax

```
#pragma push

#pragma pop
```

Remarks

The pragma `push` saves all the current pragma settings. The pragma `pop` restores all the pragma settings that resulted from the last `push` pragma. For example, see the following listing.

Listing: push and pop Example

```
#pragma peephole on
#pragma packstruct on

#pragma push        // push all compiler settings

#pragma peephole off

#pragma packstruct off

                    // pop restores "peephole" and "packstruct"

#pragma pop
```

If you are writing new code and need to set a pragma setting to its original value, use the `reset` argument, described in Using Pragmas.

This pragma does not correspond to any panel setting in the Language panel.

9.6.7 syspath_once

Controls how include files are treated.

Syntax

```
#pragma syspath_once on | off | reset
```

Remarks

If you enable this pragma, files called in `#include <>` and `#include ""` directives are treated as distinct, even if they refer to the same file.

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (syspath_once)`, described in Checking Pragma Settings. By default, this setting is enabled. For example, the same include file could reside in two distinct directories.

9.7 Library and Linking Control Pragmas

The 56800x has the following pragmas:

- define_section
- explicit_zero_data
- initializedzerodata
- section
- use_rodata

9.7.1 define_section

This pragma controls the definition of a custom section.

Syntax

```
#pragma define_section <sectname> <istring> [ <ustring> ] [ <accmode> ]
```

Remarks

```
Arguments:

<sectname>
```

Identifier by which this user-defined section is referenced in the source, that is, via the following instructions:

```
#pragma section <sectname> begin

__declspec(<sectname>)

<istring>
```

Section name string for initialized data assigned to <section>.

For example:

```
".data"

Optional Arguments:

<ustring>
```

Section name string for uninitialized data assigned to <section>. If `ustring` is not specified then `istring` is used.

`<accmode>`

One of the following indicates the attributes of the section

Table 9-2. Section Attributes

`R`	Readable
`RW (default)`	Readable and writable
`RX`	Readable and executable
`RWX`	Readable, writable, and executable

NOTE

For an example of `define_section`, see Listing: Sample Code - pragma define_section and pragma section.

Related Pragma

section

9.7.2 explicit_zero_data

Controls the section where zero-initialized global variables are emitted.

Syntax

```
#pragma explicit_zero_data on | off | reset
```

Remarks

If you enable this pragma, zero-initialized global variables are emitted to the .data section (which is normally stored in ROM) instead of the .BSS section. This results in a larger ROM image. This pragma should be enabled if customized startup code is used and it does not initialize the .BSS section. The .BSS section is initialized to zero by the default CodeWarrior startup code.

This pragma does not correspond to any setting in the Language panel. To check this setting, use `__option(explicit_zero_data)`, described in Checking Pragma Settings. By default, this pragma is disabled.

NOTE

The pragmas `explicit_zero_data` and `initializedzerodata` are the same, however, the preferred syntax is `explicit_zero_data`.

9.7.3 initializedzerodata

Controls the section where zero-initialized global variables are emitted.

Syntax

```
#pragma initializedzerodata on | off | reset
```

Remarks

If you enable this pragma, zero-initialized global variables are emitted to the .data section (which is normally stored in ROM) instead of the .BSS section. This results in a larger ROM image. This pragma should be enabled if customized startup code is used and it does not initialize the .BSS section. The .BSS section is initialized to zero by the default CodeWarrior startup code.

This pragma does not correspond to any setting in the Language panel. To check this setting, use `__option(initializedzerodata)`, described in Checking Pragma Settings. By default, this pragma is disabled.

NOTE

The pragmas `initializedzerodata` and `explicit_zero_data` are the same, however, the preferred syntax is `explicit_zero_data`.

9.7.4 section

This pragma controls the organization of object code.

Syntax

```
#pragma section <sectname> begin
  [...data..]
  #pragma section <sectname> end
```

Remarks

Argument:

`<sectname>`

Identifier by which this user-defined section is referenced in the source.

Listing: Sample Code - pragma define_section and pragma section

```
/* 1. Define the section */
#pragma define_section mysection ".mysection.data" RW

/* 2. Specify the data to be put into the section. */

#pragma section mysection begin

int a[10] = {'0','1','2','3','4','5','6','7','8','9'};

int b[10];

#pragma section mysection end

int main(void) {

    int i;

    for (i=0;i<10;i++)

      b[i]=a[i];

}

/* 3. In the linker command file, add ".mysection.data" in the ".data"
sections area of the linker command file by inserting the following
line:

    * (.mysection.data)

*/
```

RelatedPragma

define_section

9.7.5 use_rodata

Controls the section where constant data is emitted.

Compatibility

This pragma is compatible with the DSP56800, but it is not compatible with the DSP56800E.

Syntax

```
#pragma use_rodata [ on | off | reset ]
```

Remarks

By default, the compiler emits const defined data to the .data section. There are two ways to cause the compiler to emit const defined data to the .rodata section:

- Setting the "write const data to .rodata section" option in the M56800 Processor Settings panel.

  This method is a global change and emits all const-defined data to the .rodata section for the current build target.

- Using #pragma use_rodata [on | off | reset].

  on Write const data to .rodata section.

  off Write const data to .data section.

  reset Toggle pragma state.

To use this pragma, place the pragma before the const data that you wish the compiler to emit to the .rodata section. This method overrides the target setting and allows a subset of constant data to be emitted to or excluded from the .rodata section.

To see the usage of the pragma use_rodata see the code example in the following listing.

Listing: Sample Code _ Pragma use_rodata

```
const UInt16 len_l_mult_ls_data = sizeof(l_mult_ls_data) /

sizeof(Frac32) ;

const Int16 g = a+b+c;


#pragma use_rodata on
```

```
const Int16    d[]={0xdddd};

const Int16    e[]={0xeeee};

const Int16    f[]={0xffff};

#pragma use_rodata off

main()

{

           // ... code

}
```

You must then appropriately locate the .rodata section created by the compiler using the linker command file. For example, see the following listing.

Listing: Sample Linker Command FIle - Pragma use_rodata

```
MEMORY {
    .text_segment   (RWX) : ORIGIN = 0x2000, LENGTH = 0x00000000

    .data_segment   (RW)  : ORIGIN = 0x3000, LENGTH = 0x00000000

    .rodata_segment (R)   : ORIGIN = 0x5000, LENGTH = 0x00000000

}

SECTIONS {

.main_application :

    {

           # .text sections
```

```
    } > .text_segment



    .main_application_data :

    {

            # .data sections

            # .bss sections

    } > .data_segment



    .main_application_constant_data:

    {

            # constant data sections

            * (.rodata)

    } > .rodata_segment

}
```

9.8 Object Code Organization and Generation Pragmas

The 56800x has the following pragmas:

- always_inline

- auto_inline
- const_strings
- defer_codegen
- dont_inline
- dont_reuse_strings
- enumsalwaysint
- inline_bottom_up
- interrupt (for the DSP56800)
- interrupt (for the DSP56800E)
- packstruct
- pool_strings
- readonly_strings
- reverse_bitfields
- suppress_init_code
- syspath_once

9.8.1 always_inline

Controls the use of inlined functions.

Syntax

```
#pragma always_inline on | off | reset
```

Remarks

This pragma is strongly deprecated. Use the **Inline Depth** pull-down menu of the Language panel instead.

If you enable this pragma, the compiler ignores all inlining limits and attempts to `inline` all functions where it is legal to do so.

This pragma does not correspond to any panel setting. To check this setting, use `__option (always_inline)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.8.2 auto_inline

Controls which functions to inline.

Syntax

```
#pragma auto_inline on | off | reset
```

Remarks

If you enable this pragma, the compiler automatically chooses functions to inline for you.

This pragma corresponds to the **Auto-Inline** setting in the Language panel. To check this setting, use `__option (auto_inline)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.8.3 const_strings

Controls the `const`-ness of string literals.

Syntax

```
#pragma const_strings [ on | off | reset ]
```

Remarks

If you enable this pragma, the compiler will generate a warning when string literals are not declared as const. The following listing shows an example.

Listing: const_strings example

```
char *string1 = "hello";          /*OK, if const_strings is disabled*/

const char *string2 = "world";    /* Always OK */
```

This pragma does not correspond to any setting in the Language panel. To check this setting, use `__option (const_strings)`, described in Checking Pragma Settings.

9.8.4 defer_codegen

Controls the inlining of functions that are not yet compiled.

Syntax

```
#pragma defer_codegen on | off | reset
```

Remarks

This setting lets you use inline and auto-inline functions that are called before their definition:

Listing: defer_codegen Example

```
#pragma defer_codegen on
#pragma auto_inline on

extern void f();

extern void g();

main()

{

    f(); // will be inlined

    g(); // will be inlined

}

inline void f() {}

void g() {}
```

NOTE

The compiler requires more memory at compile time if you enable this pragma.

This pragma corresponds to the **Deferred Inlining** setting in the Language panel. To check this setting, use the `__option (defer_codegen)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.8.5 dont_inline

Controls the generation of inline functions.

Syntax

```
#pragma dont_inline on | off | reset
```

Remarks

If you enable this pragma, the compiler does not inline any function calls. However, it will not override those declared with the `inline` keyword. Also, it does not automatically inline functions, regardless of the setting of the `auto_inline` pragma. If you disable this pragma, the compiler expands all inline function calls, within the limits you set through other inlining-related pragmas.

This pragma corresponds to the **Don't Inline** setting of the **Inline Depth** pull-down menu of the Language panel. To check this setting, use `__option (dont_inline)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.8.6 dont_reuse_strings

Controls whether or not to store each string literal separately in the string pool.

Syntax

```
#pragma dont_reuse_strings on | off | reset
```

Remarks

If you enable this pragma, the compiler stores each string literal separately. Otherwise, the compiler stores only one copy of identical string literals. This pragma helps you save memory if your program contains a lot of identical string literals that you do not modify.

For example, take this code segment:

```
char *str1="Hello";

char *str2="Hello";


*str2 = 'Y';
```

If you enable this pragma, `str1` is `"Hello"`, and `str2` is `"Yello"`. Otherwise, both `str1` and `str2` are `"Yello"`.

This pragma corresponds to the **Reuse Strings** setting in the Language panel. To check this setting, use `__option (dont_reuse_strings)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.8.7 enumsalwaysint

Specifies the size of enumerated types.

Syntax

```
#pragma enumsalwaysint on | off | reset
```

Remarks

If you enable this pragma, the C compiler makes an enumerated type the same size as an `int`. If an enumerated constant is larger than `int`, the compiler generates an error. Otherwise, the compiler makes an enumerated type the size of any integral type. It chooses the integral type with the size that most closely matches the size of the largest enumerated constant. The type could be as small as a `char` or as large as a `long int`.

The following listing shows an example.

Listing: Example of Enumerations the Same as Size as int

```
enum SmallNumber { One = 1, Two = 2 };
  /* If you enable enumsalwaysint, this type is

     the same size as an int. Otherwise, this type is

     short int. */

enum BigNumber

  { ThreeThousandMillion = 3000000000 };

  /* If you enable enumsalwaysint, the compiler might

     generate an error. Otherwise, this type is

     the same size as a long int. */
```

This pragma corresponds to the **Enums Always Int** setting in the Language panel. To check this setting, use `__option (enumsalwaysint)`, described in Checking Pragma Settings. By default, this pragma is disabled.

NOTE

The size of a `char` on the DSP56800 target is 16 bits, and 8 bits on the DSP56800E.

9.8.8 inline_bottom_up

Controls the bottom-up function inlining method.

Syntax

```
#pragma inline_bottom_up on | off | reset
```

Remarks

Bottom-up function inlining tries to expand up to eight levels of inline leaf functions. The maximum size of an expanded inline function and the caller of an inline function can be controlled by the pragmas shown in the following two listings.

Listing: Maximum Complexity of an Inlined Function

```
//  maximum complexity of an inlined function

#pragma inline_max_size( max )          // default max == 256
```

Listing: Maximum Complexity of a Function that Calls Inlined Functions

```
//  maximum complexity of a function that calls inlined functions

#pragma inline_max_total_size( max )    // default max == 10000
```

where *max* loosely corresponds to the number of instructions in a function.

If you enable this pragma, the compiler calculates inline depth from the last function in the call chain up to the first function that starts the call chain. The number of functions the compiler inlines from the bottom depends on the values of `inline_depth`, `inline_max_size`, and `inline_max_total_size`. This method generates faster and smaller source code for some (but not all) programs with many nested inline function calls.

If you disable this pragma, top-down inlining is selected, and the `inline_depth` setting determines the limits for top-down inlining. The `inline_max_size` and `inline_max_total_size` pragmas do not affect the compiler in top-down mode.

This pragma corresponds to the **Bottom-up Inlining** setting in the Language panel. To check this setting, use `__option (inline_bottom_up)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.8.9 interrupt (for the DSP56800)

Controls the compilation of object code for interrupt service routines (ISR).

Compatibility

This pragma is compatible with the DSP56800, but it is not compatible with the DSP56800E. For the DSP56800E, see interrupt (for the DSP56800E).

Syntax

```
#pragma interrupt [called|warn|saveall[warn]]
```

Remarks

The compiler generates a special prologue and epilogue for functions so that they may be used to handle interrupts. The contents of the epilogue and prologue vary depending on the mode selected.

The compiler also emits an RTI or RTS for the return statement depending upon the mode selected. The SA, R, and CC bits of the OMR register are set to system default.

There are several ways to use this pragma as described below:

- `pragma interrupt [warn]`

  The compiler performs the following using the pragma interrupt [warn] argument:

  - Sets M01 to -1 if M01 is used by ISR
  - Sets OMR to system default (see OMR settings)
  - Saves/restores only registers used by ISR
  - Generates an RTI to return from interrupt.
  - If [warn] is present, then emits warnings if this ISR makes a function call that is not defined with `#pragma interruptcalled`

  Important considerations of usage:

  - This type of usage is required within the ISR function body as follows:

```
void ISR(void)

{

#pragma interrupt

... code here
```

- `pragma interrupt [called]`

  The compiler performs the following using the pragma interrupt [called] argument:

- Saves/restores only registers used by routine
- Generates an RTS to return from function

Important considerations of usage:

- You must use this argument before the interrupt body is compiled
- You can use this argument on the function Syntax or within the function body as described below.

On the function Syntax:

```
#pragma interrupt called

void function_called_from_interrupt (void);
```

Within the function body:

```
void function_called_from_interrupt (void)

{

#pragma interrupt called

asm (nop);

}
```

- You should use this pragma for all functions called from `#pragma interrupt` enabled ISRs. This is optional for `#pragma interrupt saveall` enabled ISRs, since for this case, the entire context is saved.

- `pragma interrupt saveall [warn]`

The compiler performs the following using the pragma interrupt saveall [warn] argument:

- Always sets M01 to -1
- Sets OMR to system default (see OMR settings)
- Saves/restores entire hardware stack via runtime call
- Generates an RTI to return from interrupt
- If [warn] is present, then emits a warning if the ISR makes a function call that is not defined with #`pragma interrupt called`

Important considerations of usage:

- This type of usage is required within the ISR function body as follows:

```
void interrupt_function(void)

{
```

```
#pragma interrupt saveall
```

... code here

- Use this pragma if the runtime library is called by the interrupt routine

In the following table, the advantages and disadvantages of the `interrupt` and `interrupt saveall` pragmas are listed.

Table 9-3. Comparision of Usage

Pragma	Advantages	Disadvantages
`interrupt saveall`	• Entire context save • No need for `#pragma interrupt called` for called functions	Larger initial performance hit due to entire context save, but becomes advantageous for ISRs with several function calls
`interrupt`	• Smaller context save, less performance hit • Generally good for ISRs with a small number of function calls	`#pragma interrupt called` required for all called functions

9.8.10 interrupt (for the DSP56800E)

This pragma controls the compilation of object code for interrupt routines.

Compatibility

This pragma is not compatible with the DSP56800, but it is compatible with the DSP56800E. For the DSP56800, see interrupt (for the DSP56800).

Syntax

```
#pragma interrupt [<options>] [<mode>] [on|off|reset]
```

Remarks

An Interrupt Service Routine (ISR) is a routine that is executed when an interrupt occurs. Setting C routines as ISRs is done using pragmas (pragma interrupt). To make a routine service an interrupt, you must:

- Write the routine.
- Set up the routine so that it is called when some interrupt occurs.

The pragma interrupt option can be used to:

- Instruct the compiler to push register values on the software stack at entry to a C function and restore them upon exit.
- Preserve the register values for the function that was interrupted.
- Emit an RTI for the return statement depending upon the mode selected. If the interrupt routine has a return value, the return register is not saved.

There are several ways to use this pragma, with an on|off|reset arguments, or with no arguments.

Table 9-4. Arguments

<options>	alignsp	Aligns the stack pointer register correctly to allow long values to be pushed on to the stack. Use this option when your project mixes C code and assembly code. Use this option specifically on ISRs which may interrupt assembly routines that do not maintain the long stack alignment requirements at all times. Restores the stack pointer to its original value before returning from the subroutine.
	comr	The Operating Mode Register (OMR) is set for the following to ensure correct execution of C code in the ISR: • 36-bit values used for condition codes.(CM bit cleared) • Convergent Rounding(R bit cleared) • No Saturation mode(SA bit cleared) • Instructions fetched from P memory.(XP bit cleared)
<mode>	saveall	Preserves register values by saving and restoring all registers by calling the INTERRUPT_SAVEALL and INTERRUPT _ RESTOREALL routines in the Runtime Library.
	called	Preserves register values by saving and restoring registers used by the routine. The routine returns with an RTS. Routines with pragma interrupt enabled in this mode are safe to be called by ISRs.
	default	This is the mode when no mode is specified. In this mode, the routine preserves register values by saving and restoring the registers that are used by the routine. The routine returns with an RTI.
	fast	56800E fast interrupt processing has lower overhead than normal interrupts processing and should be used for all

Table continues on the next page...

Table 9-4. Arguments (continued)

		low-latency time-critical interrupts. Place before any function that is a fast interrupt handler: #pragma interrupt fast.
on|off|reset	on	Enables the option to compile all C routines as interrupt routines.
	off	Disables the option to compile all C routines as interrupt routines.
	reset	Restores the option to its previous setting.

NOTE

Use on or off to change the pragma setting, and then use reset to restore the previous pragma setting.

To disable the pragma, use `#pragma interrupt off` after `#pragma interrupt`, in the following listing.

Listing: Sample Code - #pragma interrupt on | off | reset

```
#pragma interrupt off  // To be used as default value
// Non ISR code

#pragma interrupt on
void ISR_1(void) {
  // ISR_1 code goes here.
}

void ISR_2(void) {
  // ISR_2 code goes here.
}
#pragma interrupt reset
```

If the pragma is inside a function block, compile the current routine as an interrupt routine. If the pragma is not inside a function block, compile the next routine as an interrupt routine. This concept is developed in the following listing.

Listing: Sample Code - #pragma interrupt and Function Block

```
// Non ISR code
void ISR_1(void) {
#pragma interrupt
  // ISR_1 code goes here.
}
#pragma interrupt
void ISR_2(void) {
  // ISR_2 code goes here.
}
#pragma interrupt off
```

See the following listing for an example of using the 'called' option in the interrupt pragma.

Listing: Sample Code - Using the `called' Option in #pragma interrupt

```
extern long Data1, Data2, Datain;
void ISR1_inc_Data1_by_Data2(void)
{
/* This is a routine called by the interrupt service routine ISR1(). */
#pragma interrupt called
Data1+=Data2;
return;
}
void ISR1(void)
{
/* This is an interrupt service routine. */
#pragma interrupt
Data2 = Datain+2;
ISR_inc_Data1_by_Data2();
}
```

56800E fast interrupt processing has lower overhead than normal interrupts processing and should be used for all low-latency time-critical interrupts. A new compiler pragma exists and it has to be placed before any function that is a fast interrupt handler: #pragma interrupt fast.

Fast interrupt processing uses the FRTID instruction to return from the handler. The FRTID instruction has 2 delay slots in which instructions from the handler can be scheduled and the compiler performs this task automatically. If no eligible instructions are found, NOPs are inserted in these delay slots.

For the following C function:

```
#pragma interrupt fast
void foo () {
int x = 0;
}
```

The compiler generates the following assembly code:

```
clr.w X:(SP+0)
frtid
adda  #0x000002,SP
suba  #2,SP
```

Instead of:

```
clr.w X:(SP+0)
adda  #0x000002,SP
suba  #2,SP
frtid
nop
nop
```

According to 56800E core manual, there are a series of limitations for the first instructions in a fast interrupt handler. The compiler generates assembly code following these associated rules.

Consider the handler below:

```
 #pragma interrupt fast
 void foo() {
  asm (jsr 0);
  asm (move.w 1, x0);
}
```

for which the compiler generates the following sequence with 3 NOPs before the JSR instruction, according to the hardware rules for fast interrupts handlers:

```
adda  #0x000002,SP
move.wX0,X:(SP)
nop
nop
nop
jsr   0x000000
move.w#1,X0
frtid
move.wX:(SP),X0
suba  #2,SP
```

9.8.10.1 Avoiding Possible Hitches with Enabled Pragma Interrupt

If a routine that has pragma interrupt enabled (caller) calls another C function/routine (callee), it is possible that the callee can change some registers that are not saved by the caller. For example, use of volatile registers by callee.

```
//Example of a hitch

#pragma interrupt
ISR_caller()
{
      // Register B not used
      Subroutine_Callee();
}

Subroutine_Callee()
{
     // Register B used and won't be saved/restored as this is volatile register
}
```

To avoid this, use either of the following options:

- Use pragma interrupt *called* mode for the callee
- Use the pragma interrupt *saveall* mode for the caller.

```
// Solution for the hitch using called mode

#pragma interrupt
ISR_caller()
{
     // Register B not used
     Subroutine_Callee();
}

#pragma interrupt called
Subroutine_Callee()
{
     // Register B used and will be saved and restored due to use of interrupt called mode
}
```

The first option may be more efficient because only the registers that are used are preserved. The second option is easier to implement, but is likely to have a large overhead.

The situation described above also holds true for library functions because library functions do not have pragma interrupt enabled. These calls include: C Standard Library calls and Runtime Library calls (such as multiplication, division and floating point math).

9.8.11 packstruct

Controls the alignment of long words in structures.

Compatibility

This pragma is compatible with the DSP56800, but it is not compatible with the DSP56800E.

Syntax

```
#pragma packstruct on | off | reset
```

Remarks

If you enable this pragma, `integer longs` within structures are aligned on four byte boundaries. When this pragma is disabled there is no alignment within structures. This pragma does not correspond to any setting in the Language panel. To check this setting, `use __option(packstruct)`, described in Checking Pragma Settings. By default, this pragma is enabled.

9.8.12 pool_strings

Controls how the compiler stores string constants.

Compatibility

This pragma is not compatible with the DSP56800, but it is compatible with the DSP56800E.

Syntax

```
#pragma pool_strings on | off | reset
```

Remarks

If you enable this setting, the compiler collects all string constants into a single data object so that your program needs only one TOC entry for all of them. While this decreases the number of TOC entries in your program, it also increases your program size because it uses a less efficient method to store the address of the string.

If you disable this setting, the compiler creates a unique data object and TOC entry for each string constant.

Enable this setting if your program is large and has many string constants.

The **Pool Strings** setting corresponds to the pragma poolstring. To check this setting, use `__option (pool_strings)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.8.13 readonly_strings

Controls the output of C strings to the read only data section.

Syntax

```
#pragma readonly_strings on | off | reset
```

Remarks

If you enable this pragma, C strings used in your source code (for example, " `hello`") are output to the read-only data section (.rodata) instead of the global data section (.data). In effect, these strings act like `const char *`, even though their type is really `char *`.

For the DSP56800, this pragma corresponds to the "Make Strings Read Only" panel setting in the **M56800 Processor** settings panel. To check this setting, use `__option (readonly_strings)`, described in Checking Pragma Settings.

For the DSP56800E, there is no "Make Strings Read Only" panel setting in the **M56800E Processor** settings panel.

9.8.14 reverse_bitfields

Controls whether or not the compiler reverses the bitfield allocation.

Syntax

```
#pragma reverse_bitfields on | off | reset
```

Remarks

This pragma reverses the bitfield allocation.

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (reverse_bitfields)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.8.15 suppress_init_code

Controls the suppression of static initialization object code.

Syntax

```
#pragma suppress_init_code on | off | reset
```

Remarks

If you enable this pragma, the compiler does not generate any code for static data initialization.

Warning

Using this pragma can produce erratic or unpredictable behavior in your program.

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (suppress_init_code)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.8.16 syspath_once

Controls how include files are treated.

Syntax

```
#pragma syspath_once on | off | reset
```

Remarks

If you enable this pragma, files called in `#include <>` and `#include ""` directives are treated as distinct, even if they refer to the same file.

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (syspath_once)`, described in Checking Pragma Settings. By default, this setting is enabled. For example, the same include file could reside in two distinct directories.

C Standard Library and Runtime Library (CW libraries) functions require the AGU (Address Generation Unit) to be in linear addressing mode, that is, the M01 registers are set to -1. If a function is interrupted and was using modulo address arithmetic, any calls to CW libraries from the ISR do not work unless the M01 is set to -1 in the ISR. Also, the M01 register would need to be restored before exiting the ISR so that the interrupted function can resume as before, with the same modulo address arithmetic mode settings.

9.9 Optimization Pragmas

The 56800x has the following pragmas:

- div_nonstd32by16_canoverflow
- factor1
- factor2
- factor3
- nofactor1
- nofactor2
- nofactor3
- opt_common_subs
- opt_dead_assignments
- opt_dead_code
- opt_lifetimes
- opt_loop_invariants
- opt_propagation
- opt_strength_reduction
- opt_strength_reduction_strict
- opt_unroll_loops
- optimization_level
- optimize_for_size
- peephole

9.9.1 div_nonstd32by16_canoverflow

Enables the `F_idiv_ls_canoverflow` and `F_idiv_uls_canoverflow` runtime functions.

Syntax

```
#pragma div_nonstd32by16_canoverflow on
```

Remarks

The high-performance, specialized integer 32-bit by 16-bit runtime functions `F_idiv_ls_canoverflow` and `F_idiv_uls_canoverflow` are intended for use in applications. The calling code ensures that the division result of a 32-bit numerator and a 16-bit denominator fits into a 16-bit field and does not overflow. These functions reduce execution time. Use this pragma to generate calls to these functions.

9.9.2 factor1

Turns on factorization step 1.

Syntax

```
#pragma factor1
```

Remarks

Compiler performs the factorization step 1. To turn off factor1, use nofactor1. This optimization is performed on global variables before register allocation, takes into account register pressure, and replaces absolute addressing with indirect addressing.

This pragma does not correspond to any panel setting in the Language panel. By default, this pragma is enabled at global optimization level 2 and above.

9.9.3 factor2

Turns on factorization step 2.

Syntax

```
#pragma factor2
```

Remarks

Compiler performs the factorization step 2. To turn off factor2, use nofactor2.This optimization is performed on global variables after register allocation, replaces absolute addressing with indirect addressing, and detects a physical address register that is available to do the factorization. Register allocation spilling decreases pressure so new webs, that could not be created before register allocation, can be created.

This pragma does not correspond to any panel setting in the Language panel. By default, this pragma is enabled at global optimization level 2 and above.

9.9.4 factor3

Turns on factorization step 3.

Syntax

```
#pragma factor3
```

Remarks

Compiler performs the factorization step 3. To turn off factor3, use nofactor3. This optimization is performed on local variables after register allocation. (SP-offset) addressing is transformed in register indirect addressing. This optimization is performed after register allocation because only at this point are the local variables accessed by stack location.

This pragma does not correspond to any panel setting in the Language panel. By default, this pragma is enabled at global optimization level 2 and above.

9.9.5 nofactor1

Turns off factorization step 1.

Syntax

```
#pragma nofactor1
```

Remarks

Compiler does not perform the factorization step 1. To turn on factorization step 1, use factor1.

This pragma does not correspond to any panel setting in the Language panel.

9.9.6 nofactor2

Turns off factorization step 2.

Syntax

```
#pragma nofactor2
```

Remarks

Compiler does not perform the factorization step 2. To turn on factorization step 2, use factor2.

This pragma does not correspond to any panel setting in the Language panel.

9.9.7 nofactor3

Turns off factorization step 3.

Syntax

```
#pragma nofactor3
```

Remarks

Compiler does not perform the factorization step 3. To turn on factorization step 3, use factor3.

This pragma does not correspond to any panel setting in the Language panel.

9.9.8 opt_common_subs

Controls the use of common subexpression optimization.

Syntax

```
#pragma opt_common_subs on | off | reset
```

Remarks

If you enable this pragma, the compiler replaces similar redundant expressions with a single expression. For example, if two statements in a function both use the expression

```
a * b * c + 10
```

the compiler generates object code that computes the expression only once and applies the resulting value to both statements.

The compiler applies this optimization to its own internal representation of the object code it produces.

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (opt_common_subs)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.9.9 opt_dead_assignments

Controls the use of dead store optimization.

Syntax

```
#pragma opt_dead_assignments on | off | reset
```

Remarks

If you enable this pragma, the compiler removes assignments to unused variables before reassigning them.

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (opt_dead_assignments)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.9.10 opt_dead_code

Controls the use of dead code optimization.

Syntax

```
#pragma opt_dead_code on | off | reset
```

Remarks

If you enable this pragma, the compiler removes a statement that other statements never execute or call.

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (opt_dead_code)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.9.11 opt_lifetimes

Controls the use of lifetime analysis optimization.

Syntax

```
#pragma opt_lifetimes on | off | reset
```

Remarks

If you enable this pragma, the compiler uses the same processor register for different variables that exist in the same routine but not in the same statement.

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (opt_lifetimes)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.9.12 opt_loop_invariants

Controls the use of loop invariant optimization.

Syntax

```
#pragma opt_loop_invariants on | off | reset
```

Remarks

If you enable this pragma, the compiler moves all computations that do not change inside a loop outside the loop, which then runs faster.

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (opt_loop_invariants)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.9.13 opt_propagation

Controls the use of copy and constant propagation optimization.

Syntax

```
#pragma opt_propagation on | off | reset
```

Remarks

If you enable this pragma, the compiler replaces multiple occurrences of one variable with a single occurrence.

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (opt_propagation)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.9.14 opt_strength_reduction

Controls the use of strength reduction optimization.

Syntax

```
#pragma opt_strength_reduction on | off | reset
```

Remarks

If you enable this pragma, the compiler replaces array element arithmetic instructions with pointer arithmetic instructions to make loops faster.

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (opt_strength_reduction)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.9.15 opt_strength_reduction_strict

Uses a safer variation of strength reduction optimization.

Syntax

```
#pragma opt_strength_reduction_strict on | off | reset
```

Remarks

Like the opt_strength_reduction pragma, this setting replaces multiplication instructions that are inside loops with addition instructions to speed up the loops. However, unlike the regular strength reduction optimization, this variation ensures that the optimization is only applied when the array element arithmetic is not of an unsigned type that is smaller than a pointer type.

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (opt_strength_reduction_strict)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.9.16 opt_unroll_loops

Controls the use of loop unrolling optimization.

Syntax

```
#pragma opt_unroll_loops on | off | reset
```

Remarks

If you enable this pragma, the compiler places multiple copies of a loop's statements inside a loop to improve its speed.

This pragma does not correspond to any panel setting in the Language panel. To check this setting, use `__option (opt_unroll_loops)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.9.17 optimization_level

Controls global optimization.

Syntax

```
#pragma optimization_level 0 | 1 | 2 | 3 | 4
```

Remarks

This pragma specifies the degree of optimization that the global optimizer performs.

To select optimizations, use the pragma `optimization_level` with an argument from `0` to `4`. The higher the argument, the more optimizations performed by the global optimizer.

These pragmas correspond to the settings in the **Global Optimizations** panel. By default, this pragma is disabled.

9.9.18 optimize_for_size

Controls optimization to reduce the size of object code.

Syntax

```
#pragma optimize_for_size on | off | reset
```

Remarks

This setting lets you choose what the compiler does when it must decide between creating small code or fast code. If you enable this pragma, the compiler creates smaller object code at the expense of speed. This pragma does not effect the inline directive or the inlining of explicitly inlined functions. This pragma can be used in conjunction with the dont_inline pragma to decrease the code size. If you disable this pragma, the compiler creates faster object code at the expense of size.

The pragma corresponds to the **Optimize for Size** setting on the **Global Optimizations** panel. To check this setting, use `__option (optimize_for_size)`, described in Checking Pragma Settings. By default, this pragma is disabled.

9.9.19 peephole

Controls the use peephole optimization.

Syntax

```
#pragma peephole on | off | reset
```

Remarks

If you enable this pragma, the compiler performs *peephole optimizations*, which are small, local optimizations that eliminate some compare instructions and improve branch sequences.

For the DSP56800, this pragma corresponds to the **Peephole Optimization** setting in the M56800 Processor settings panel. Yet for the DSP56800E, there is no corresponding setting for the M56800 Processor settings panel. To check this setting, use `__option (peephole)`, described in Checking Pragma Settings.

9.10 Profiler Pragmas

The 56800x has just one profiler pragma:

- profile

9.10.1 profile

Controls code to enable or disable the profiler.

Syntax

```
#pragma profile on | off | reset
```

Remarks

This setting lets you choose whether the compiler adds code to a function to call profiler library functions. If you enable this pragma, the compiler calls profiling functions at the beginning and end of the current function. If you disable this pragma, the compiler adds no additional code. For further information on the profiler, see the Chapter "Profiler" in either of the Targeting Manuals.

The pragma corresponds to the **Generate code for profiling** setting on the **M56800E Processor** settings panel. To check this setting, use `__option (profile)`, described in Checking Pragma Settings. By default, this pragma is disabled.

Chapter 10
Predefined Symbols

The compiler preprocessor has predefined macros that describe the compile-time environment and properties of the target processor.

This chapter describes how to use these predefined symbols and lists them:

10.1 Using Predefined Symbols

Predefined symbols are in the preprocessor, available at compile-time only.

10.2 Version Symbol

Version symbols:

10.2.1 __MWERKS__

Defined with the version of the CodeWarrior compiler.

CodeWarrior compilers issued after 1995 define this macro with the compiler's version. For example, if the compiler version is 3.2, the value of `__MWERKS__` is `0x3200`.

This macro is defined as `1` if the compiler was issued before the CodeWarrior CW7 that was released in 1995.

10.3 Date and Time Symbol

Date and time symbol:

- __DATE__
- __TIME__

10.3.1 __DATE__

Defined as the date during compilation.

During compilation, the compiler defines this macro with a character string representation of the current date.

10.3.2 __TIME__

Defined as the time of day during compilation.

During compilation, the compiler defines this macro with a character string representation of the current time.

10.4 Name Symbols

Name symbols:

- __FILE__
- __LINE__

10.4.1 __FILE__

The name of the source code file being compiled.

During compilation, the compiler defines this macro with a character string representation of the name of the file being compiled.

10.4.2 __LINE__

The number of the line of source code being compiled.

During compilation, this macro is defined as an integer value representing the number of line of source code being compiled.

10.5 Object Code Organization and Generation Symbol

Object code organization and generation symbol:

- __m56800E__
- __profile__
- __optlevelx

10.5.1 __m56800E__

This preprocessor macro is defined always from the m56800E compiler.

Syntax

```
__m56800E__
```

Remarks

This macro is defined by default.

10.5.2 __profile__

Defined as 1 when generating object code that works with a profiler. Undefined otherwise.

10.5.3 __optlevelx

Optimization level exported as a predefined macro.

Syntax

```
__optlevel0
__optlevel1
__optlevel2
__optlevel3
__optlevel4
```

Remarks

Using these macros, user can conditionally compile code for a particular optimization level. The following table lists the level of optimization provided by the `__optlevelx` macro.

Table 10-1. Optimization Levels

Macro	Optimization Level
`__optlevel0`	O0
`__optlevel1`	O1
`__optlevel2`	O2
`__optlevel3`	O3
`__optlevel4`	O4

Example

The listing below shows an example of `__optlevelx` macro usage.

Listing: Example usage of __optlevel macro

```
int main()
{
#if __optlevel0
```

```
... // This code compiles only if this code compiled with Optimization
level 0
#elif __optlevel1
... // This code compiles only if this code compiled with Optimization
level 1
#elif __optlevel2
... // This code compiles only if this code compiled with Optimization
level 2
#elif __optlevel3
... // This code compiles only if this code compiled with Optimization
level 3
#elif __optlevel4
... // This code compiles only if this code compiled with Optimization
level 4
#endif
}
```

10.6 C Symbols

C symbol:

- __STDC__

10.6.1 __STDC__

Defined as 1 when compiling ISO Standard C source code, undefined otherwise.

The compiler defines this macro as 1 when the compiler's settings are configured to restrict the compiler to translate source code that conforms to the ISO C Standard. The compiler does not define this macro otherwise.

Chapter 11
Optimization

CodeWarrior build tools offer features to reduce the size of object code, improve a program's execution speed, and often do both at the same time. Compiler optimizations rearrange, add, or remove instructions to reduce size or improve performance.

This chapter describes how to take advantage of these optimizations:

11.1 Optimization Considerations

There are several issues to take into consideration when selecting optimizations. Code can be optimized for size or for speed, and there are optimizations that could effect the size and the performance of the compiler. It is important to understand the full effects of the optimizations. For example, inlining will decrease the overhead of making function calls. However, if too many functions are called the resulting executable could be too large to run on the target platform.

Inlining also effects the ability to debug a program. Programs are optimally debugged at optimization level 0, and with no additional optimization options enabled. Users should keep in mind that optimization could result in incorrect data being displayed while debugging, and stepping through functions could also seem incorrect.

Finally, the performance of the compiler could also be negatively effected by enabling optimizations. If there are many optimizations enabled, the compile time could increase because of the extra time needed to process the optimizations.

All of these issues should be considered when selecting optimizations.

11.2 Inlining

When inlining is enabled certain function calls are replaced with the function code. Inlining function optimizes for speed, as there is no call. However, overall code may be larger if function code is repeated in several places.

The inlining of a function is based on the complexity of the function and the settings of several compiler options: IPA, Inline Depth, Auto Inline and Bottom up inline.

11.3 Profiling

For more details about profiling, see the *CodeWarrior Development Studio for Microcontrollers Version10.x Profiling and Analysis Tools User Guide*.

11.4 String Literals

The compiler and linker manage character strings so that they occupy less space in the object code and executable file.

String literals are:

- Pooling Strings
- Reusing Strings

11.4.1 Pooling Strings

The **Pool Strings** setting in the C/C++ Language Panel controls how the compiler stores string constants.

If you enable this setting, the compiler collects all string constants into a single data object so that your program needs only one TOC (table of content) entry for all of them. While this decreases the number of TOC entries in your program, it also increases your program size because it uses a less efficient method to store the address of the string.

If you disable this setting, the compiler creates a unique data object and TOC entry for each string constant.

Enable this setting if your program is large and has many string constants.

The **Pool Strings** setting corresponds to the pragma pool_strings. To check this setting, use __option (pool_strings). By default, this setting is disabled. See also pool_strings and Checking Pragma Settings.

11.4.2 Reusing Strings

The **Reuse Strings** setting in the C/C++ Language Panel controls how the compiler stores string literals.

If you enable this setting, the compiler stores each string literal separately. Otherwise, the compiler stores only one copy of identical string literals. This means if you change one of the strings, you change them all. For example, look at this code:

```
char *str1="Hello";

char *str2="Hello"; // two identical strings

*str2 = 'Y';
```

This setting helps you save memory if your program contains identical string literals which you do not modify. If you enable the Reuse Strings setting, the strings are stored separately. After changing the first character, str1 is still `Hello`, but str2 is `Yello`.

If you disable the Reuse Strings setting, the two strings are stored in one memory location because they are identical. After changing the first character, both str1 and str2 are `Yello`, which is counterintuitive and can create bugs that are difficult to locate. The Reuse Strings setting corresponds to the pragma dont_reuse_strings. To check this setting, use __option (dont_reuse_strings). By default, this setting is enabled, so strings are not reused. See also dont_reuse_strings and Checking Pragma Settings.

11.5 Optimizations

The following is a collection of optimization types and examples of how the resulting generated code is affected:

- Dead Code Elimination

- Expression Simplification
- Common Subexpression Elimination
- Copy Propagation
- Dead Store Elimination
- Live Range Splitting
- Loop-Invariant Code Motion
- Strength Reduction
- Loop Unrolling
- M56800E Specific Optimizations

11.5.1 Dead Code Elimination

Listing: Dead Code Elimination, Before Optimization

```
void func(void)
{
 if (0)
 {
    otherfunc1();
 }
 otherfunc2();
}
```

Listing: Dead Code Elimination, After Optimization

```
void func_optimized(void)
{
  otherfunc2();
}
```

11.5.2 Expression Simplification

Listing: Expression Simplification, Before Optimization

```
#define MY_OFFSET 4

void func(int* result1, int* result2, int* result3, int* result4, int x)
{
```

```
 *result1 = x + 0;
 *result2 = x * 2;
  *result3 = x - x;
  *result4 = 1 + x + MY_OFFSET;
}
```

Listing: Expression Simplification, After Optimization

```
#define MY_OFFSET 4

void func_optimized(int* result1, int* result2, int* result3, int* result4, int x)
{
 *result1 = x;
 *result2 = x << 2;
 *result3 = 0;
 *result4 = 5 + x;
}
```

11.5.3 Common Subexpression Elimination

Listing: Common Subexpression Elimination, Before Optimization

```
void func(int* vec, int size, int x, int y, int value)

{
    if (x * y < size)
    {
        vec[x * y] = value;
    }
}
```

Listing: Common Subexpression Elimination, After Optimization

```
void func_optimized(int* vec, int size, int x, int y, int value)

{
    int temp;
    temp = x * y;
    if (temp < size)
    {
        vec[temp] = value;
    }
}
```

11.5.4 Copy Propagation

Listing: Copy Propagation, Before Optimization

```
void func(int* a, int x)
{
   int i;
   int j;
   j = x;
   for (i = 0; i < j; i++)
   {
```

```
        a[i] = j;
    }
}
```

Listing: Copy Propagation, After Optimization

```
void func_optimized(int* a, int x)
{
    int i;
    int j;
    j = x;
    for (i = 0; i < x; i++)
    {
        a[i] = x;
    }
}
```

11.5.5 Dead Store Elimination

Listing: Dead Store Elimination, Before Optimization

```
void func(int x, int y)
{
 x = y * y;
 otherfunc1(y);
 x = getresult();
 otherfunc2(y);
}
```

Listing: Dead Store Elimination, After Optimization

```
void func_optimized(int x, int y)
{
 otherfunc1(y);
 x = getresult();
 otherfunc2(y);
}
```

11.5.6 Live Range Splitting

Listing: Live Range Splitting, Before Optimization

```
void func(int x, int y)
{

 int a;

 int b;

 int c;

 a = x * y;

 otherfunc(a);
```

```
 b = x + y;

 otherfunc(b);

 c = x - y;

 otherfunc(c);

}
```

Listing: Live Range Splitting, After Optimization

```
void func_optimized(int x, int y)

{

 int temp;

 temp = x * y;

 otherfunc(temp);

 temp = x + y;

 otherfunc(temp);

 temp = x - y;

 otherfunc(temp);

}
```

11.5.7 Loop-Invariant Code Motion

Listing: Loop-Invariant Code Motion, Before Optimization

```
void func(float* vec, int max, float val)
{
  float circ;
  int i;
  for (i = 0; i < max; ++i)
  {
     circ = val * 2 * PI;
     vec[i] = circ;
  }
}
```

Listing: Loop-Invariant Code Motion, After Optimization

```
void func_optimized(float* , int max, float val)
{
 float circ;
 int i;
 circ = val * 2 * PI;
 for (i = 0; i < max; ++i)
 {
    vec[i] = circ;
 }
}
```

11.5.8 Strength Reduction

Listing: Strength Reduction, Before Optimization

```
void func(int* vec, int max, int fac)
{
   int i;
    for (i = 0; i < max; ++i)
    {
      vec[i] = fac * i;
    }
}
```

Listing: Strength Reduction, After Optimization

```
void func_optimized(int* vec, int max, int fac)
{
   int i;
   int temp = 0;
   for (i = 0; i < max; ++i)
   {
      vec[i] = temp;
      temp = temp + fac;
   }
}
```

11.5.9 Loop Unrolling

Listing: Loop Unrolling, Before Optimization

```
const int MAX = 100;

void func(int* vec)

{
   int i;
   for (i = 0; i < MAX; ++i)
    {
       otherfunc(vec[i]);
    }
}
```

Listing: Loop Unrolling, After Optimization

```
const int MAX = 100;
void func_optimized(int* vec)
{
    int i;
    for (i = 0; i < MAX;)
    {
       otherfunc(vec[i]);
       ++i;
       otherfunc(vec[i]);
```

```
        ++i;
    }
}
```

11.5.10 M56800E Specific Optimizations

This section provides techniques, programming style suggestions, and information to maximize the efficiency of the Freescale C compiler for the 56800/E Digital Signal controllers.

11.5.10.1 Overview of the 56800E Architecture

The 56800/E processors are member of the 56800x family of digital signal microcontrollers. The 56800x instruction set is targeted for efficient microcontroller code generation and DSP (Digital Signal Processing). The 56800/E are digital signal processors, because they both have a microcontroller and DSP.

Microcontroller instructions include:

- Bit manipulation instructions
- Flexible branching instructions
- Absolute (global) addressing modes to maximize control code density.

DSP features include:

- Single cycle MAC (Multiply-Accumulate)
- Separate address register file
- Separate data/program memory spaces,
- Multiple addressing modes, including pointer post-update addressing modes.

The C compiler attempts to target the post-update addressing modes in loops. In this chapter, we describe the programming style that promotes the selection of the post-update addressing modes.

The 56800x family is a native 16-bit machine - data and addresses are 16 bits wide. The 56800/E extends the address bus width to 24-bits (called the large data model), allowing a wider range of data addresses, but at a cost of performance and code density. In this chapter, we discuss the techniques used to minimize the cost of enabling the large data model.

NOTE

Although ANSI-C data types are fully supported, in this chapter, we show that the best code is generated when the programmer favors the native data type size (16-bits).

11.5.10.2 Working with the 56800E Memory Models

The Freescale 56800E C Compiler supports large and small program and data memory models as shown in the following table. The small data model is more code efficient. However, sometimes the application requires a larger data address space.

Table 11-1. Code and Data Memory Ranges

Section	Small Data Model		Large Data Model	
	Size (KB)	Range (Word Address)	Size (MB)	Range (Word Address)
CODE (P:memory)	128	0 - 0xFFFF	1	0 - 0x7FFFF
DATA (X:memory)	128	0 - 0xFFFF	32	0 - 0x7FFFF
DATA (X:memory) character data	64	0 - 0xFFFF	16	0 - 0x7FFFF

The large data memory model allows data to be placed in memory at addresses greater than the 16-bit address limitation of the small data model. The large data memory model can be selected at **DSC Compiler > Processor Options** in the **Tool Settings** panel. This selection informs the compiler that global and static data should be addressed with the 24-bit variants of the absolute addressing modes of the device. Also in the large memory model, pointers are treated as 24-bit quantities when moved from register to register, memory to register, or register to memory. For information on how the large memory model is selected, see the *Freescale 56800/E Hybrid Controllers: MC56F83xx/ DSP5685x Family Targeting Manual*.

One likely scenario in an embedded programming environment is that the total static and global data size, that is, the total size of data objects that the compiler accesses with absolute addressing modes (X:xxxx or X:xxxxxx addressing modes) will comfortably reside within the 16-bit data addressing range. However, the heap (dynamically allocated data memory) or the stack (local, automatic data memory) may require extended addressing as this data may extend beyond the 16-bit address range.

To optimize the program size, use the CodeWarrior IDE targets settings panel **M56800E Processor: Large Data Model: Globals live in lower memory** panel option in conjunction with the large data memory model. The **Globals live in lower memory** panel option reverts the absolute addressing modes to the small data model for static and global

variables, while using the large memory model for any address pointers or local variables. Thus, for static and global variables, the efficiency of the small data model is retained even for programs where the total data size may exceed the 16-bit addressing range.

The following listing shows the code generation differences between the large and small data model. In this example, the code performs a bubble sort on an array of integers. At maximum optimization, the code runs in 579 cycles in the small data memory model. The code takes 760 cycles using the large data memory model. When the large data memory model and **Globals live in lower memory** option is selected, the code runs in 729 cycles. The difference in the cycle count of the two large data model runs is due to the way global variables are addressed. The **Globals live in lower memory** option forces the access of the global variable "next" to be there as it would be for the small data model.

Listing: Example 1: Memory Model Comparison Code

```
int vector[] = { 3,7,6,1,2,5 }; int next; int main()
{
int i=0, j=0;
int sz = sizeof(vector)/sizeof(int);
     for (i=0; i<sz; i++){
          for(j=0; j<sz-i; j++){
               if (vector[j]>vector[j+1]) {
                    next=vector[j];
                    vector[j]=vector[j+1];
                    vector[j+1]=next;
                    }
               }
          }
     }
}
```

Table 11-2. Example 1 at Maximum Optimization

Small Data Model	Large Data Model	Large Data Model and Global Live in Lower Memory
579 cycles	760 cycles	729 cycles

If the **Globals live in lower memory** option is selected, be sure to locate the `.data` and `.bss` sections in lower memory. Dynamically allocated memory and the stack may be located in either lower or upper memory for the large data model.

11.5.10.3 Targeting Post-Update Addressing Modes in Loops

Post-update addressing modes are available for many 56800E instructions. At optimization level 2 and above, the compiler attempts to locate register-based address expressions which change by a linear amount for each iteration through a loop. If such an expression is located and certain conditions are met, the compiler may replace the address update expression with a post-update addressing mode that is performed concurrently with the move or arithmetic operation. Such a transformation is called `strength reduction' in compiler terminology and means replacing an instruction operation with a cheaper (fewer cycles or words) instruction. Address expressions are normally either address registers that have been loaded directly with the addresses of objects (variables) or address registers holding the calculated address of array elements. Array indices which vary by a regular, linear amount for each iteration through a loop are called `induction variables.' Many times induction variables are completely eliminated when their function is replaced by a post-update addressing mode.

Listing: Example 2: Post-Update Addressing Modes

```
X:(Rn)+         Address is incremented by 1 (2 for move.l)

X:(Rn)-         Address is decremented by 1 (2 for move.l)

X:(Rn)+N        Address is incremented by value in N register
```

Some programming guidelines which promote the successful targeting of the post-update addressing mode are:

- The address expression must be within a loop.
- The address expressions must be register based, therefore, global pointer variables are usually not targeted for strength reduction since they may be accessed with absolute addressing modes. Sometimes, it is useful to load the address of a global array into a local pointer variable to make the address expression more obvious to the compiler.
- The address expression should be executed each iteration of the loop. Address expressions embedded in `if-then-else' blocks will not be targeted for post-update addressing.
- Induction variables must be defined at one point in the loop and must vary linearly from its previous value.

In the following listing, a simple loop that calculates the sum of elements in a local array is shown. For this example, the induction variable `i' is completely eliminated because:

- A DO loop instruction has been generated, eliminating the need for a test on `i' to determine if the loop has ended
- The use of `i' in the calculation of the array addresses has been eliminated, in favor of a post-update addressing mode (see line 11 in the following listing)

Listing: Example 3: Successful Strength Reduction

```
     int i;
     int sum=0;
     int arr[] = { 13,14,18,3,7,0,1,4,11,20 };
     int sz = sizeof(arr)/sizeof(int);
     for (i=0; i < sz; i++)
          sum += arr[i];
     printf ( "Sum is %d\n",sum );
Assembly output:
(1) adda #<10,SP              ;allocate stack
(2) move.w #<0,B                ;sum = 0
(3) adda #-9,SP,R1            ;&arr[0]->R1
(4) moveu.w #F47,R0              ;temp F47->R0
(5) do #<10,>_L8_0          ;compiler generated init loop
(6) move.w X:(R0)+,A            ;initialize arr[]
(7) move.w A1,X:(R1)+
(8)_L8_0:
(9) adda #-9,SP,R0            ;&arr[0]->R0
(10) do #<10,>_L8_1         ;for loop
(11) move.w X:(R0)+,A           ;arr[i]->A
(12) add A,B                 ;sum = arr[i]+sum
(13)_L8_1:
(14) adda #<2,SP              ;printf call setup
(15) moveu.w #@lb(F54),N         ;string temp to stack
(16) move.w N,X:(SP)
(17) move.w B1,X:(SP-1)         ;sum to stack
(18) jsr >Fprintf            ;call printf
(19) suba #<2,SP              ;restore stack
```

The following listing shows a case where strength reduction of the address expression was not possible, mainly because the access to the array is conditionally executed in the loop. Also, the induction variable `i' is used in the `if' test, but this would not normally prevent a post-update transformation from occurring.

Listing: Example 4: Array Update In Conditional Block

```
for (i=0; i < sz; i++)

    if ( i & 1 )

      sum += arr[i];

Assembly output:

 (1) do          #<10,>_L8_1     ;for loop

 (2) brclr       #1,Y0,<_L8_2    ;if ( i & 1 )

 (3) move.w      X:(R0),A        ;arr[i]->Av

 (4) add         A,B             ;sum = arr[i]+sum

 (5)_L8_2:

 (6) adda        #<1,R0          ; &arr = &arr + 1;

 (7) add.w       #<1,Y0          ; i = i + 1

 (8) nop

 (9)_L8_1:
```

In the following listing, another situation is shown where strength reduction will fail to find a post-update opportunity. This is when the loop or induction variable is multiply defined in a loop.

NOTE

This also kills the hardware do loop as the compiler cannot determine the static loop count.

Listing: Example 5: Induction Variable is Multiply Defined

```
     for (i=0; i < sz; i++)

           sum += arr[i++];

Assembly output:

 (1) move.w      #<0,A               ; i=0

 (2)_L8_1:

 (3) move.w      A1,B                ; i -> temp

 (4) add.w       #<1,B               ; temp++

 (5) move.w      A1,N                ; temp++ -> N

 (6) adda        #-9,SP,R0           ; &arr[0] -> R0

 (7) move.w      X:(R0+N),A          ; arr[temp++] -> A
```

```
 (8) add         A,Y0                ; sum = arr[i++] + sum
 (9) move.w      B1,A                    ; temp++ -> i
 (10)add.w       #<1,A               ; i = i + 1
 (11)cmp.w       #<10,A
 (12)blt         <_L8_1              ; i < 10 ?
```

The following listing demonstrates a simple delay line loop that is structured so post-update addressing is impossible. The final store to memory in the loop is a memory plus displacement addressing mode, move.w A1,X:(R0+1), which doesn't allow post-update addressing. The loop written as is takes approximately 29 cycles and 9 words for NTAPS=6.

Listing: Example 6: Loop Structure Doesn't Allow Post-Update Addressing

```
for (ii = NTAPS - 2; ii >= 0; ii--) {
      z[ii + 1] = z[ii];
}
Assembly output:
 (1) do #<5,>_L12_1        ; for ()
 (2) move.w Y0,R0             ; ii -> R0
 (3) adda R3,R0             ; &z[0] + i
 (4) move.w X:(R0),A          ; z[ii] -> A
 (5) move.w A1,X:(R0+1)       ; z[ii] -> z[ii + 1]
 (6) sub.w #<1,Y0             ; ii--
 (7)_L12_1:
```

The loop in the above listing may be re-written slightly as shown in the following listing to allow for much more efficient processing. The idea is to try to get an instruction that has a post-update variant as the final load or store in the loop. This loop executes in 17 cycles and 8 words.

Listing: Example 7: Loop Re-Written to Allow Post-Update Addressing

```
     int *p1 = &z[NTAPS-1];
     for (ii = NTAPS - 2; ii >= 0; ii--) {
     *p1-- = z[ii];
     }
Assembly output:
```

```
(1) tfra R1,R3                ;&z[NTAPS-1] -> R3
(2) adda #-5,SP,R0             ;&z[NTAPS-2] -> R0
(3) tfra R0,R2                ;R0 -> R2
(4) do #<5,>_L9_1            ;for ()
(5) move.w X:(R2)-,B              ;z[ii] -> B
(6) move.w B1,X:(R3)-             ;B -> z[ii+1]
(7)_L9_1:
```

11.5.10.3.1 The Effects of Casting on Code Quality

The 56800x family is a native 16-bit architecture. Type casting to and from 16-bit data types requires extra instruction words and cycles. Use 16-bit types (int, short, unsigned int, unsigned short) whenever possible to minimize to program memory required for the application. Also be aware that ANSI-C requires implicit promotion of integral types for arithmetic operations and this may cause implicit type casting. Of course, favoring 16-bit data types may cause an increase in the total data size of an application. The trade off between program and data memory will have to be judged for each application. In general, if program memory is the limiting resource, favor 16-bit types. If data memory is the limiting resource, then using 8-bit data types where possible may be preferred.

Casting ints to char or long types are usually the least costly in terms of words and cycles. Since accumulators (A,B,C,D registers in the 56800E) are the only registers capable of holding 32-bit quantities, they must be used for long operations. Accumulators are composed of two individually addressable 16-bit parts, the MSP or most significant portion and the LSP or least significant portion. The MSP is often treated as a 16-bit register containing an int or short sized quantity (16-bits). An int to long cast requires an asr16 instruction to move the MSP to the LSP of the accumulator.

Listing: Example 8: Casting an integer to a Long Data Type

```
int ls;
long ll;
    ll = (long)ls;
move.w X:(SP-2),A;
asr16 A,A
move.l A10,X:(SP-4)
```

Bytes or `char` variables are stored as portions of `integer` sized registers. The 56800E does not contain 8-bit registers. An int to char cast requires an explicit sign extension (sxt.b) of the integer to properly format the register so that the sign bit of the char is extended into the entire word. This is required for proper arithmetic operations on the char since arithmetic in C occurs on integers by definition. Also, the 56800E only performs 16-bit and 32-bit arithmetic.

Listing: Example 9: Casting an int to a char Data Type

```
  char lc;
  int ls;

  lc = (char)ls;

Assembly output:
move.w X:(SP-2),A
sxt.b A,A
move.b A1,X:(SP)
```

`Chars` that are converted to `int` or `long` first require a sign extension of the byte into an integer value. If the `char` is converted to a long, an addition asr16 is required to convert to a 32-bit value.

Listing: Example 10: Casting a char to long

```
  long ll;
  char lc;

  ll = (long)lc;

Assembly output:
moveu.b X:(SP),A
sxt.b A,A
asr16 A,A
move.l A10,X:(SP-4)
```

It should be clear now that casting causes runtime penalties in terms of code size and cycles. Sometimes the perceived benefit of using shorter data types to save data memory results in runtime costs.

The 56800E has a unique model for handling pointers to character data. Although the data memory is organized by words, that is, each address points to a word (two bytes) of data, individual bytes within a word can be still be addressed. The compiler handles this addressing invisibly, but the programmer should be aware of the costs of converting from byte pointers to word pointers and vice versa.

A byte address is generated by the compiler when the programmer chooses to use character data to represent an object. Strings are character data by default in the 56800E compiler and are addressed with byte pointers. Special instructions in the 56800E instruction set expect to see and operate on byte pointer values. A word pointer may be converted to a byte pointer by multiplying the word address by two. Similarly, a byte address is converted to a word address by dividing the byte address by two. When a byte pointer is cast to a word pointer, an explicit, runtime conversion of the pointer quantity is performed. The cost is a one word, one cycle penalty to bit shift the address value to the left, that is, multiply by two, to convert to a byte pointer. The cost is the same to convert to a word pointer, except the shift is to the right, effectively dividing by two. The void pointer is a byte pointer since the void pointer should be able to represent any data type, including chars. Since there is a runtime penalty for converting pointer types, casts back

and forth should be limited for efficient C programs. This may be a factor when the void pointer is used to point to generic data and cast to the proper type at runtime. The following listing shows the effect of casting byte and word pointers.

Listing: Example 11: Casting Byte and Word Pointers

```
  void * pvoid;
  int vint;
  int * pint;
  char *pchar;

  pint = (int *)&vint;
adda #-5,SP,R0
move.w R0,X:(SP-6)

  pvoid = (void *)pint;
moveu.w X:(SP-6),R0
asla R0,R0
move.w R0,X:(SP-4)

  pchar = (char *)pint;
move.w X:(SP-6),R0
asla R0,R0
move.w R0,X:(SP-7)

  pint = (int *)pvoid;
moveu.w X:(SP-4),R0
lsra R0
move.w R0,X:(SP-6)
```

11.5.10.3.2 Miscellaneous Techniques

There are other several minor techniques to be aware of when writing the most efficient C code for the compiler.

Initialize local arrays and structures at declaration time, if possible. Local arrays and structures are initialized optimally by the compiler.

Functions with a large number of parameters will probably have to pass some parameters on the stack causing costly memory accesses. Make sure that frequently called functions pass their parameters in registers. For information on the parameter passing rules for the 56800E C Compiler see the *Freescale 56800/E Hybrid Controllers: MC56F83xx/ DSP5685x Family Targeting Manual*.

Forcing enums as integers (C/C++ Language Panel, "Enums Always Ints") may yield better code since integers are usually handled more efficiently.

Loading frequently used global variables into local temporary variables sometimes has a positive effect on code size and performance, since accessing variables through registers is more efficient that absolute addressing modes.

As an illustration of the final point in the list above, the code in the following listing executes in 98 cycles and 20 program memory words. The same function is performed by the code in Listing: Example 13: Modified Global Structure Example, but it executes in

57 cycles and 13 program memory words. A temporary local variable is used in processing instead of the global variable. Fewer absolute addressing instructions account for the difference.

Listing: Example 12: Global Structure Example

```
#define ARRAY_SIZE 5
static struct s1
{
     unsigned char value_a;
     unsigned char value_b;
     unsigned char value_c;
} s_s1[ARRAY_SIZE];
unsigned int r1;
int main()
{
     int i;
     for (i = 0; i < ARRAY_SIZE; i++)
     {
     r1 += s_s1[i].value_a;
     r1 += s_s1[i].value_b;
     r1 += s_s1[i].value_c;
     }
     return (r1);
}
```

Listing: Example 13: Modified Global Structure Example

```
int main() {
      int i;
      unsigned int local_var;
      local_var = r1;
      for (i = 0; i < ARRAY_SIZE; i++)
      {
            local_var += s_s1[i].value_a;
            local_var += s_s1[i].value_b;
            local_var += s_s1[i].value_c;
```

```
    }


    r1 = local_var;

    return (r1);

}
```

11.5.10.4 Software Pipelining

Software pipelining is a loop transformation that changes the initial loop so that parts of different iterations execute at the same time. This scheduling technique exploits architectural instruction level parallelism.

It may also produce better loop schedules when stalls, hazards or latencies exist between instructions in the initial loop, if they can be avoided in the transformed loop.

Note that the DSP56800e architecture provides limited parallelism by means of parallel move instructions. These limitations narrow down the applicability of this transformation.

An example of software pipelining transformation:

```
#include "intrinsics_56800e.h"
int x[100], y[100], i;
long res;
void main()
{
    long t=0;
    for (i=0; i<100; i++)
    {
        t = L_mac(t, x[i], y[i]);
    }
    res = t;
}
```

This code will compile the loop-body into one cycle:

```
  rep         R1
  mac         Y0,X0,A         X:(R0)+,Y0        X:(R3)+,X0
```

where mac instruction from first iteration of the loop executes in parallel with load instructions from the second iteration of the initial loop.

This transformation applies to the inner most loops of a program, and currently is enabled only for DO loops.

It is controlled by the `-[no]swp` command line switch, and it is by default enabled for optimization levels higher than 2. Otherwise `#pragma swplevel on/off` may be used to control the transformation. When optimizing for size, software pipelining is disabled, as it usually increases program size.

11.5.10.5 Stack Sequence Optimization

This transformation replaces several accesses to adjacent stack locations with a post-increment/-decrement addressing mode by using an available address register.

For DSP56800E, this transformation may bring performance gain both in execution speed and code size. Speed is improved as instructions using post-increment access usually take only one cycle as opposed to instructions with immediate offsets that can take 2 or 3 cycles. Code size is reduced when large immediates are present.

An example of stack sequence optimization where the following low-level intermediate piece of code:

```
move.w      X:(SP-2),A

move.w      X:(SP-1),Y1

move.w      X:(SP-2),A

move.w      X:(SP-3),B

add.w       X:(SP-4),B
```

will become:

```
adda        #-2,SP,R0

move.w      X:(R0)+,A

move.w      X:(R0)-,Y1

move.w      X:(R0)-,A

move.w      X:(R0)+,B

add.w       X:(R0),B
```

which brings an improvement of 3 cycles (2+2+2+2+3 as opposed to 2+1+1+1+1+2).

In the example above, the transformation actually increases the code size, and that is why it will not be performed on this example when -Os optimization is required.

Note that this transformation makes use of both post increment and post decrement update modes, and it can also exploit all instructions accessing the stack, not only loads and stores.

Transformation is controlled by the `-[no]stackseq` command line switch, and it is enabled by default for an optimization level higher than one. Also, `#pragma stackseq on/off` may be used to control the transformation.

11.5.10.6 Constant to Array Reallocation

Constants/large constants encoded in instructions are stored into an array in data memory and immediate operands are changed into data memory access using register-indirect, post-increment operands.

The main target of this optimization is speed, but occasionally size improvements can also be obtained.

Each transformed instruction reduces the execution time of an instruction by 1-2 cycles and reduces program memory size by 1-2 words, but also causes an increase of data memory by 1-2 words, depending on the size of immediates.

Besides the operand mode transformation, grouping transformed instructions can further decrease total program memory size.

The following instructions take between 2-3 words of program memory and 2-3 cycles to execute:

```
MOVE.W#xxxx, HHHHH
MOVE.L#xxxxxx, HHHHH
```

and they are transformed to:

```
MOVE.W(Rx)+, HHHHH
MOVE.L(Rx)+, HHHHH
```

so that the resulting instruction will take 1word of program memory and 1 cycle to execute, but it will add an extra 1-2 words into data memory (the immediate values). It will also add an overhead of one instruction per sequence for computing the address of the first element.

If no instruction grouping happens with instructions transformed to post-increment indirect addressing, the total memory size used will slightly increase, due to the computation of stack offset for the first element in a sequence.

An example of how this optimization works on the following piece of low-level intermediate code:

```
.code
    move.w  X:(R3)+, X0
    move.w  #<number_1>, Y0
    mac     Y0, X0, A
    move.w  X:(R3)+, X0
    move.w  #<number_2>, Y0
    mac     Y0, X0, A
```

The code above can be optimized to:

```
.code
     move.w  #<array_starting_address>, R0
     move.w  X:(R3)+, X0
     move.w  X:(R0)+, Y0
     mac     Y0, X0, A
     move.w  X:(R3)+, X0
     move.w  X:(R0)+, Y0
     mac     Y0, X0, A


 .data
      array_starting_address:
       <number_1>
       <number_2>
```

This optimization is disabled for `-Os` and is automatically enabled on speed optimization level >= 2. Constant to array reallocation can be enabled/disabled at any optimization level using `-[no]constarray` options in the command line. At function level, you should use `#pragma constarray on/off`.

NOTE

This optimization creates extra data for its own use in the `.data` section. If the `.data` section load address is different from the `.data` section run address (e.g., because of an AT linker command file directive) avoid using the constant to array optimization in any functions executing before the run address. Use `#pragma constarray off` to ensure proper function execution.

11.5.10.7 Interprocedural Analysis Support

Interprocedural Analysis (IPA) allows the compiler to generate better and/or smaller code by inspecting more than just one function or data object at the same time. This technology is currently used by the inliner.

The compiler supports three different interprocedural analysis modes: off (default), file, and program.

With the function mode `-ipa off`, functions are optimized and code is generated when the function has been parsed. This mode allows no interprocedural analysis.

With the mode `-ipa file`, a translation unit is completely parsed before any code or data is generated. This allows optimizations and inlining on a per-file basis. This mode will require more memory and it can be slightly slower than the `-ipa off` mode. The compiler will also do an early dead code/data analysis in this mode, so objects with internal linkage that are not referenced will be dead-stripped in the compiler rather than in the linker.

With the mode `-ipa program` all translation units are completely parsed. Optimizations and code generation are done in a final stage enabling true "whole program" optimizations. For example, auto-inlining of functions that are defined in another translation unit.

"Program IPA" can require a lot of memory and will also be slower, especially in the change/build/debug cycle because all code generation and optimizations will have to be redone whenever a program has to be relinked.

Using this mode from command-line tools is more complicated. If you specify all source files on the command-line you can use `-ipa program`:

```
mwcc56800e -ipa program test1.c test2.c [all sources and libraries]...
```

This will compile, optimize, codegen, and link binary in "program" ipa mode.

If you want to separate compilation from linking you can either use:

```
mwcc56800e -ipa program -c test1.c
```

This generates test1.o file (empty) and a test1.irobj file.

```
mwcc56800e -ipa program -c test2.c
```

This generates test2.o file (empty) and a test2.irobj file.

```
mwcc56800e -ipa program test1.o test2.o [all *.o and libraries]...
```

This will optimize, codegen, and link binary in "program" ipa mode.

If you want to invoke the linker separately you will have to use:

```
mwcc56800e -ipa program -c test1.c
```

This generates test1.o file (empty) and a test1.irobj file.

```
mwcc56800e -ipa program -c test2.c
```

This generates test2.o file (empty) and a test2.irobj file.

```
mwcc56800e -ipa program-final test1.irobj test2.irobj [all *.irobjs]...
```

This will optimize and codegen in "program" ipa mode and update the .o files.

```
mwld56800e -o test.exe test1.obj test2.obj [all *.objs and libraries]...
```

This will link binaries.

The `.irobj` files contain an intermediate program representation. Thus the build step corresponding to "make clean" should remove these when the matching `.o` file is deleted.

NOTE

-ipa program mode is available only with command line compiler.

WARNING

-ipa program mode is not fully-tested for DSC development. Use at your own risk.

Chapter 12
Tool Performance

CodeWarrior compilers can "precompile" a header file to speed up translation of source code. Precompiling a header file that is included often in other source files will reduce the time the compiler uses to translate source code.

Some options for CodeWarrior compilers and linkers affect how much time these tools use. By managing these options so that they are used only when they are needed, you can reduce the time needed to build your software.

12.1 Precompiled Header Files

This topic contains the following sub-topics:

12.1.1 When to Use Precompiled Files

Source code files in a project typically use many header files. Typically, the same header files are included by each source code file in a project, forcing the compiler to read these same header files repeatedly during compilation. To shorten the time spent compiling and recompiling the same header files, CodeWarrior compilers can precompile a header file, allowing it to be subsequently preprocessed much faster than a regular text source code file.

For example, as a convenience, programmers often create a header file that contains commonly-used preprocessor definitions and includes frequently-used header files. This header file is then included by each source code file in the project, saving the programmer some time and effort while writing source code.

This convenience comes at a cost, though. While the programmer saves time typing, the compiler does extra work, preprocessing and compiling this header file each time it compiles a source code file that includes it.

This header file can be precompiled so that, instead of preprocessing multiple duplications, the compiler needs to load just one precompiled header file.

12.1.2 What Can be Precompiled

A file to be precompiled does not have to be a header file (`.h` or `.hpp` files, for example), but it must meet these requirements:

- The file must be a source code file in text format.

  You cannot precompile libraries or other binary files.

- A C source code file that will be automatically precompiled must have `.pch` file name extension.
- Precompiled files must have a `.mch` file name extension.
- The file to be precompiled does not have to be in a CodeWarrior IDE project, although a project must be open to precompile the file.

  The CodeWarrior IDE uses the build target settings to precompile a file.

- The file must not contain any statements that generate data or executable code.

  However, the file may define static data.

- Precompiled header files for different build targets are not interchangeable.
- A source file may include only one precompiled file.
- A file may not define any items before including a precompiled file.

  Typically, a source code file includes a precompiled header file before anything else (except comments).

12.1.3 Precompiling C++ Source Code

The CodeWarrior C++ compiler has these requirements for precompiling source code:

- C source code may not include precompiled C++ header files and C++ source code may not include precompiled C header files.
- C++ source code can contain inline functions and constant variable declarations (`const`)
- A C++ source code file that will be automatically precompiled must have a `.pch++` file name extension.

12.1.4 Using a Precompiled Header File

Although a precompiled file is not a text file, you use it like you would a regular header file. To include a precompiled header file in a source code file, use the `#include` directive.

NOTE

Unlike regular header files in text format, a source code file may include only one precompiled file.

Tip

Instead of explicitly including a precompiled file in each source code file with the `#include` directive, put the `#include` directive in the **Prefix Text** field of the **C/C++ Preprocessor** settings panel and make sure that the **Use prefix in precompiled headers** option is on. If the **Prefix File** field already specifies a file name, include the precompiled file in the prefix file with the `#include` directive.

The following listing shows an example.

Listing: Header File that Creates a Precompiled Header File for C

```
// sock_header.pch

// When compiled or precompiled, this file will generate a

// precompiled file named "sock_precomp.mch"

#pragma precompile_target "sock_precomp.mch"

#define SOCK_VERSION "SockSorter 2.0"

#include "sock_std.h"

#include "sock_string.h"

#include "sock_sorter.h"
```

The following listing shows another example.

Listing: Using a Precompiled File

```
// sock_main.c
```

```
// Instead of including all the files included in
// sock_header.pch, we use sock_precomp.h instead.
//
// A precompiled file must be included before anything else.
#include "sock_precomp.mch"
int main(void)
{
  // ...
  return 0;
}
```

12.1.5 Preprocessing and Precompiling

When precompiling a header file, the compiler preprocesses the file too. In other words, a precompiled file is preprocessed in the context of its precompilation, not in the context of its compilation.

The preprocessor also tracks macros used to guard `#include` files to reduce parsing time. Thus, if a file's contents are surrounded with:

```
#ifndef FOO_H
#define FOO_H
// file contents
#endif
```

The compiler will not load the file twice, saving some small amount of time in the process.

12.1.6 Pragma Scope in Precompiled Files

Pragma settings inside a precompiled file affect only the source code within that file. The pragma settings for an item declared in a precompiled header file (such as data or a function) are saved then restored when the precompiled header file is included.

For example, the source code in the following listing specifies that the variable `xxx` is a `far` variable.

Listing: Pragma Settings in a Precompiled Header

```
// my_pch.pch

// Generate a precompiled header named pch.mch.

#pragma precompile_target "my_pch.mch"

#pragma far_data on

extern int xxx;
```

The source code in the following listing includes the precompiled version of the above listing.

Listing: Pragma Settings in an Included Precompiled File

```
// test.c

#pragma far_data off // far data is disabled

#include "my_pch.mch"   // this precompiled file sets far_data on

// far_data is still off but xxx is still a far variable
```

The pragma setting in the precompiled file is active within the precompiled file, even though the source file including the precompiled file has a different setting.

12.1.7 Precompiling a File in the CodeWarrior IDE

To precompile a file in the CodeWarrior IDE, use the **Precompile** command in the **Project** menu:

1. Start the CodeWarrior IDE.
2. Open or create a project.
3. Choose or create a build target in the project.

   The settings in the project's active build target will be used when preprocessing and precompiling the file you want to precompile.

4. Open the source code file to precompile.

   See What Can be Precompiled for information on what a precompiled file may contain.

5. From the **Project** menu, choose **Precompile**.

   A save dialog box appears.

6. Choose a location and type a name for the new precompiled file.

   The IDE precompiles the file and saves it.

7. Click **Save**.

The save dialog box closes, and the IDE precompiles the file you opened, saving it in the folder you specified, giving it the name you specified.

You may now include the new precompiled file in source code files.

12.1.8 Updating a Precompiled File Automatically

Use the CodeWarrior IDE's project manager to update a precompiled header automatically. The IDE creates a precompiled file from a source code file during a compile, update, or make operation if the source code file meets these criteria:

- The text file name ends with .pch (for C header files).
- The file is in a project's build target.
- The file uses the `precompile_target` pragma.
- The file, or files it depends on, have been modified.

The IDE uses the build target's settings to preprocess and precompile files.

Chapter 13
Libraries and Runtime Code

You can use a variety of libraries with the CodeWarrior™ IDE. The libraries include ANSI-standard libraries for C, runtime libraries, and other codes. This chapter explains how to use these libraries for DSP56800E development.

With respect to the Main Standard Library (MSL) for C, this chapter is an extension of the *MSL C Reference*. Refer the *MSL C Reference* manual for general details on the standard libraries and their functions.

This chapter includes the following sections:

13.1 MSL for DSP56800E

This section explains MSL that has been modified for use with DSP56800E. The compiler library supports C++ support functions, including trigonometric, hyperbolic, power, absolute value functions, exponential, and logarithmic functions.

NOTE

To use double precision function versions, you must use libraries that support `long long` and double data types. Libraries that support these types have names that include _SLLD. Use `#pragma slld` on to compile the project.

NOTE

Libraries are available that are precompiled for speed (the library name contains the specifier `o4p`) or precompiled for code size (the library name contains the specifier `o4s`).

13.1.1 Using MSL for DSP56800E

CodeWarrior™ Development Studio for 56800/E Digital Signal Controllers includes a version of MSL. MSL is a complete C library for use in embedded projects. All of the sources necessary to build MSL are included in CodeWarrior™ Development Studio for 56800/E Digital Signal Controllers, along with the project files for different configurations of MSL. If you already have a version of the CodeWarrior IDE installed on your computer, the CodeWarrior installer adds the new files needed for building versions of MSL for DSP56800E.

The project directory for the DSP56800E MSL is: `CodeWarrior\M56800E Support\msl\MSL_C\DSP_56800E\projects\MSL C 56800E.mcp`.

Do not modify any of the source files included with MSL. If you need to make changes based on your memory configuration, make changes to the runtime libraries.

Ensure that you include one or more of the header files located in the following directory:

`CodeWarrior\M56800E Support\msl\MSL_C\DSP_56800E\inc`

When you add the relative-to-compiler path to your project, the appropriate MSL and runtime files will be found by your project. If you create your project from Stationery, the new project will have the proper support access path.

13.1.1.1 Console and File I/O

DSP56800E Support provides standard C calls for I/O functionality with full ANSI/ISO standard I/O support with host machine console and file I/O for debugging sessions (Host I/O) through the JTAG port or HSST in addition to such standard C calls such as memory functions `malloc()` and `free()`.

A minimal “thin” printf via “console_write” and “fflush_console” is provided in addition to standard I/O.

See the *MSL C Reference* manual (Main Standard Library).

13.1.1.1.1 Library Configurations

There are Large Data Model and Small Data Model versions of all libraries. (Small Program Model default is off for all library and Stationery targets.)

MSL provides standard C library support.

The Runtime libraries provide the target-specific low-level functions below the high-level MSL functions. There are two types of Runtime libraries:

- JTAG-based Host I/O
- HSST-based Host I/O

  For each project requiring standard C library support, a matched pair of MSL and Runtime libraries are required (SDM or LDM pairs).

  The HSST library is added to HSST client-to-client DSP56800E targets.

NOTE

DSP56800E stationery creates new projects with LDM and SDM targets and the appropriate libraries.

Below is a list of the DSP56800E libraries:

- Main Standard Libraries
  - `MSL C 56800E.lib`

    Standard C library support for Small Data Model.

  - `MSL C 56800E lmm.lib`

    Standard C library support for Large Data Model.

- Runtime Libraries
  - `runtime 56800E.lib`

    Low-level functions for MSL support for Small Data Model with Host I/O via JTAG port.

  - `runtime 56800E lmm.lib`

    Low-level functions for MSL support for Large Data Model with Host I/O via JTAG port.

  - `runtime_hsst_56800E.lib`

    Low-level functions for MSL support for Small Data Model with Host I/O via HSST.

  - `runtime_hsst_56800E_lmm.lib`

    Low-level functions for MSL support for Large Data Model with Host I/O via HSST.

13.1.1.1.2 Host File Location

Files are created with `fopen` on the host machine as shown in the table below:

Table 13-1. Host File Creation Location

fopen Filename Parameter	Host Creation Location
filename with no path	target project file folder
full path	location of full path

13.1.2 Allocating Stacks and Heaps for DSP56800E

Stationery linker command files (LCF) define heap, stack, and bss locations. LCFs are specific to each target board. When you use M56800E stationery to create a new project, CodeWarrior automatically adds the LCF to the new project.

See ELF Linker for general LCF information. See each specific target LCF in Stationery for specific LCF information.

See the following table for the variables defined in each Stationery LCF.

Table 13-2. LCF Variables and Address

Variables	Address
_stack_addr	Start address of the stack
_heap_size	Size of the heap
_heap_addr	Start address of the heap
_heap_end	End address of the heap
_bss_start	Start address of memory reserved for uninitialized variables
_bss_end	End address of bss

To change the locations of these default values, modify the linker command file in your DSP56800E project.

NOTE

Ensure that the stack and heap memories reside in data memory.

13.1.2.1 Definitions

The following definitions are used throughout this document:

- Stack

- Heap
- BSS

13.1.2.1.1 Stack

The stack is a last-in-first-out (LIFO) data structure. Items are pushed on the stack and popped off the stack. The most recently added item is on top of the stack. Previously added items are under the top, the oldest item at the bottom. The “top” of the stack may be in low memory or high memory, depending on stack design and use. M56800E uses a 16- bit-wide stack.

13.1.2.1.2 Heap

Heap is an area of memory reserved for temporary dynamic memory allocation and access. MSL uses this space to provide heap operations such as malloc. M56800E does not have an operating system (OS), but MSL effectively synthesizes some OS services such as heap operations.

13.1.2.1.3 BSS

BSS is the memory space reserved for uninitialized data. The compiler will put all uninitialized data here. If the Zero initialized globals live in data instead of BSS checkbox in the M56800E Processor Panel is checked, the globals that are initialized to zero reside in the .data section instead of the .bss section. The stationery init code zeroes this area at startup. See the M56852 init (startup) code in this chapter for general information and the stationery init code files for specific target implementation details.

NOTE

Instead of accessing the original Stationery files themselves (in the Stationery folder), create a new project using Stationery which will make copies of the specific target board files such as the LCF.

13.2 Runtime Initialization

The default `init` function is the bootstrap or glue code that sets up the DSP56800E environment before your code executes. This function is in the `init` file for each board-specific stationery project. The routines defined in the `init` file performs other tasks such as clearing the hardware stack, creating an interrupt table, and retrieving the stack start and exception handler addresses.

The final task performed by the `init` function is to call the `main()` function.

The starting point for a program is set in the **Entry Point** field in the DSC Linker > Input settings panel.

The project for the DSP56800E runtime is: `CodeWarrior\M56800E Support\runtime_56800E \projects\Runtime 56800E.mcp`

Table 13-3. Library Names and Locations

Library Name	Location
Large Memory Model `Runtime 56800E lmm.lib`	CodeWarrior\M56800E Support\runtime_56800E\lib
Small Memory Model `Runtime 56800E.Lib`	CodeWarrior\M56800E Support\runtime_56800E\lib

When creating a project from R1.1 or later Stationery, the associated init code is specific to the DSP56800E board. See the startup folder in the new project folder for the init code.

Listing: Sample Initialization File (DSP56852EVM)

```
#


; ---------------------------------------------------------
;
;       56852_init.asm


;       sample
        description:  main entry point to C code.
;                 setup runtime for C and call main
;
; ---------------------------------------------------------



;===============================
; OMR mode bits
;===============================
NL_MODE                 EQU             $8000
CM_MODE                 EQU             $0100
```

```
XP_MODE                 EQU             $0080
R_MODE                  EQU             $0020
SA_MODE                 EQU             $0010

        section rtlib

        XREF F_stack_addr
        org p:


        GLOBAL Finit_M56852_

        SUBROUTINE "Finit_M56852_",Finit_M56852_,Finit_M56852END-Finit_M56852_

Finit_M56852_:

;
; setup the OMr with the values required by C
;
      bfset   #NL_MODE,omr     ; ensure NL=1  (enables nsted DO loops)
        nop
        nop
   bfclr   #(CM_MODE|XP_MODE|R_MODE|SA_MODE),omr ; ensure CM=0 (optional for C)
                   ; ensure XP=0 to enable harvard architecture
                   ; ensure R=0  (required for C)
                   ; ensure SA=0 (required for C)

; Setup the m01 register for linear addressing
        move.w  #-1,x0
        moveu.w x0,m01       ; Set the m register to linear addressing

        moveu.w hws,la        ; Clear the hardware stack
        moveu.w hws,la       nop
        nop
```

```
CALLMAIN:                           ; Initialize compiler environment



;Initialize the Stack
        move.l #>>F_Lstack_addr,r0
        bftsth #$0001,r0
        bcc noinc
        adda #1,r0
noinc:
        tfra    r0,sp                ; set stack pointer too
        move.w  #0,r1
        nop
        move.w  r1,x:(sp)
        adda    #1,sp


  jsr                   F__init_sections


; Call main()
        move.w  #0,y0               ; Pass parameters to main()
        move.w  #0,R2
        move.w  #0,R3


        jsr             Fmain       ; Call the Users program
;
;  The fflush calls where removed because they added code
;  growth in cases where the user is not using any debugger IO.
;  Users should now make these calls at the end of main if they use debugger IO
;
;       move.w  #0,r2
;       jsr     Ffflush             ; Flush File IO
;       jsr     Ffflush_console     ; Flush Console IO


;       end of program; halt CPU
        debughlt
```

```
        rts
Finit_M56852END:


       endsec
```

Index

A

C

D

E

F

G

I

K

L

M

N

O

P

R

S

T

U

V

W

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number CWMCUDSCCMPREF
Revision 10.6, 02/2014